Lua

Quick Reference

Mitchell

II

The Lua C API

C API Introduction

Lua itself is just a C library. Its three header files provide the host application with a simple API for creating an embedded Lua interpreter, interacting with it, and then closing it. Example 22 demonstrates a very basic stand-alone Lua interpreter whose command line accepts only a Lua script to run.

NOTE

The C examples in this book make use of some C99-specific features, so adapting those examples on a platform without a C99-compliant compiler will likely be necessary. However, Lua itself is written in ISO (ANSI) C, and will compile without modification.

Example 22. Simple stand-alone Lua interpreter

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main(int argc, char **argv) {
  int status = 0;
  // Create a new embedded Lua interpreter.
  lua_State *L = luaL_newstate();
  // Load all of Lua's standard library modules.
  luaL_openlibs(L);
  // Execute the Lua script specified on the command
  // line. If there is an error, report it.
  if (argc > 1 && luaL_dofile(L, argv[1]) != LUA_OK†) {
    const char *errmsg = lua_tostring(L, -1);
    fprintf(stderr, "Lua error: %s\n", errmsg);
    status = 1;
  }
  // Close the Lua interpreter.
  lua_close(L);
  return status;
}
```

The header file *lua.h* provides Lua's basic C API. All functions and macros in that file start with the prefix "`lua_`". The file *lauxlib.h* provides a higher-level API with convenience func-

† `LUA_OK` exists only in Lua 5.2 and 5.3. Lua 5.1 uses the constant 0 instead.

tions for common tasks that involve the basic API. All functions and macros in that file start with the prefix "`luaL_`". The file *lualib.h* provides Lua's standard library module API. Table 13 lists the contents of *lualib.h* for hosts that prefer to load only specific Lua standard library modules rather than all of them at once.

This book refers to Lua's API functions and macros as "API functions" for the sake of simplicity.

CAUTION

Programming with Lua in C does not make programming in C any easier. Type-checking is mandatory, memory allocation errors are possible, and segmentation faults are nearly a given when passing improper arguments to Lua's API functions. Also, any unexpected errors raised by Lua will likely cause the host program to abort. (The section "Error Handling" on page 122 describes how to avoid that unhappy scenario.)

Table 13. Standard library module API (lualib.h)

Standard Library Module Name	C Function
`""`	luaopen_base
`LUA_BITLIBNAME` ("bit32")[a]	luaopen_bit32[a]
`LUA_LOADLIBNAME` ("package")	luaopen_package
`LUA_MATHLIBNAME` ("math")	luaopen_math
`LUA_STRLIBNAME` ("string")	luaopen_string
`LUA_UTF8LIBNAME` ("utf8")[b]	luaopen_utf8[b]
`LUA_TABLIBNAME` ("table")	luaopen_table
`LUA_COLIBNAME` ("coroutine")	luaopen_coroutine[c]
`LUA_IOLIBNAME` ("io")	luaopen_io
`LUA_OSLIBNAME` ("os")	luaopen_os
`LUA_DBLIBNAME` ("debug")	luaopen_debug

[a] Only in Lua 5.2.

[b] Only in Lua 5.3.

[c] Only in Lua 5.2 and 5.3. Lua 5.1's coroutine library module is included in luaopen_base.

`lua_State`
: A C `struct` that represents both a thread in a Lua interpreter and the interpreter itself. Data can be shared between Lua threads but not between Lua interpreters.

TIP

Lua is fully re-entrant and can be used in multi-threaded code provided the macros `lua_lock` and `lua_unlock` are defined when compiling Lua.

`lua_State *luaL_newstate();`
: Returns a newly created Lua interpreter, which is also that interpreter's main thread.

`void luaL_openlibs(lua_State *L);`
: Loads all of Lua's standard library modules into Lua interpreter *`L`*.

`luaL_requiref(L, name, f, 1), lua_pop(L, 1);` **Lua 5.2, 5.3**
`lua_pushcfunction(L, f), lua_pushstring(L, name), lua_call(L, 1, 0);` **Lua 5.1**
: Loads one of Lua's standard library modules into Lua interpreter *`L`*. *`name`* is the string name of the module to load and *`f`* is that module's C function. Table 13 lists Lua's standard library module names and their associated C functions.

  Using this in place of `luaL_openlibs()` is useful for hosts that want control over which of Lua's standard library modules are available. For example, a host can prevent Lua code from interacting with the underlying operating system via the `os` module by simply not loading that module.

`void lua_close(lua_State *L);`
: Destroys, garbage collects, and frees the memory used by all values in Lua interpreter *`L`*.

The Stack

The primary method of communication between Lua and its host is through Lua's *stack*, which is treated as a "Last In, First Out" (LIFO) type of data structure. (The host however has complete access to all elements on Lua's stack and can manipulate them at will.) This book uses the term "the stack" to refer to the current Lua interpreter's stack. Communication between the host and Lua typically proceeds as follows:

1. The host pushes some C values onto the stack as Lua values.
2. The host invokes Lua to perform an operation on those values, such as defining a global variable, calling a function with arguments, or manipulating a table's contents. (During such an operation, Lua may call back into C via C functions, which are described in the section "C Functions" on page 108.)
3. The host retrieves any resulting Lua values from the stack as C values and then pops those Lua values off the stack or, if an error occurred, the host handles it gracefully.

Each element on the stack refers to a Lua value that was pushed onto it (either directly by the host or indirectly by Lua or its API functions during an operation). Also, each stack element has an *index*. Stack indices counting from the bottom of the stack are positive and start at 1, while stack indices counting from the top are negative and start at -1.

The stack has a finite size, so stack overflows are possible due to negligence. Ensuring that every value pushed onto the stack is eventually popped off helps maintain consistency. Where applicable, this book explicitly states how many stack values an API function pushes and pops.

The following sections describe how to prevent a stack overflow, how to work with stack indices, and how to push, pop, query, and retrieve stack values.

Increase Stack Size

Lua's initial stack size is 40 elements, though it is configurable when compiling Lua. The stack does not grow automatically as values are pushed onto it, so the host needs to grow it as necessary prior to pushing values in order to prevent a stack overflow.

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

Ensures the stack has room for pushing at least *n* more values onto it and returns `1` or, if the stack could not be grown any further, returns `0`.

NOTE

The maximum stack size is 8,000 elements in Lua 5.1 and 15,000 elements in Lua 5.2 and 5.3. This arbitrary limit is configurable when compiling Lua. It is possible to run out of memory before hitting the maximum stack size, especially in embedded environments.

Work with Stack Indices

The host can convert between relative and absolute stack indices, and retrieve or define the index of the stack top.

`int lua_absindex(lua_State *L, int index);` **Lua 5.2, 5.3**

Returns relative (negative) stack index *index* converted to an absolute (positive) index.

`int lua_gettop(lua_State *L);`

Returns the stack index of the value at the top of the stack, which is also the number of values currently on the stack.

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

Makes the value at stack index *index* the value at the top of the stack, filling in any empty space with nil values and popping off any extra values. The stack has exactly `lua_absindex(L, index)` values on it after this operation.

The stack size cannot be shrunk.

Push Values

The host can push various types of C values onto the stack as Lua values. The means for doing so are broken up into sections that cover how to push values of each of Lua's eight types: nil, boolean, number, string, function, table, thread, and userdata.

Push a nil

The host can push nil values onto the stack.

`void lua_pushnil(lua_State *L);`
: Pushes the value `nil` onto the stack.

Push a boolean

The host can push boolean values onto the stack.

`void lua_pushboolean(lua_State *L, int b);`
: Pushes boolean value *b* onto the stack.

Push a number

The host can push number values onto the stack. Lua provides some C type definitions that differentiate between integers and floats, since Lua numbers can be either.

`lua_Integer`
: The C type associated with Lua integers (typically `long long` in Lua 5.3 and `ptrdiff_t` in Lua 5.1 and 5.2). This is configurable when compiling Lua.

`lua_Unsigned` **Lua 5.2, 5.3**
: The C type associated with unsigned Lua integers (typically `unsigned long long` in Lua 5.3 and `unsigned long` in Lua 5.2). This is configurable when compiling Lua.

`lua_Number`
: The C type associated with Lua floats (typically `double`). This is configurable when compiling Lua.

```
void lua_pushinteger (lua_State *L, lua_Integer i);
void lua_pushunsigned(lua_State *L, lua_Unsigned i);   Lua 5.2
void lua_pushnumber  (lua_State *L, lua_Number n);
```
: Pushes onto the stack integer value *i* or float value *n*.

Push a string

The host can push various kinds of string values onto the stack. Strings can be C-style strings, strings with embedded zeros, and formatted strings. Table 14 lists the placeholders available for formatted strings.

Table 14. String formatting placeholders

Placeholder	Argument Type	Meaning
%c	`int`	Character byte
%d	`int`	Integer
%f	`lua_Number`	Float
%I[a] (upper-case 'i')	`lua_Integer`	Integer
%p	Pointer	Hexadecimal address
%s	Zero-terminated string	String
%U[a]	`long int`	UTF-8 character
%%	N/A	Literal '%'

[a] Only in Lua 5.3.

```
const char *lua_pushstring  (lua_State *L,
                             const char *s);        Lua 5.2, 5.3
void        lua_pushstring  (lua_State *L,
                             const char *s);            Lua 5.1
const char *lua_pushlstring (lua_State *L, const char *s,
                             size_t len);           Lua 5.2, 5.3
void        lua_pushlstring (lua_State *L, const char *s,
                             size_t len);               Lua 5.1
const char *lua_pushliteral (lua_State *L,
                             "literal");            Lua 5.2, 5.3
void        lua_pushliteral (lua_State *L, "literal"); Lua 5.1
const char *lua_pushfstring (lua_State *L,
                             const char *format, ...);
const char *lua_pushvfstring(lua_State *L,
                             const char *format,
                             va_list argp);
```

Pushes onto the stack zero-terminated string value *`s`*, string value *`s`* of length *`len`* bytes, a literal string value, formatted string value constructed from both string *`format`* and a variable number of arguments, or formatted string value constructed from both string *`format`* and variable argument list *`argp`*. Returns a pointer to Lua's internal copy of the string.

`format` contains a sequence of placeholders that specify how to format their respective arguments. Table 14 lists valid placeholders along with their meanings.

NOTE

Lua makes an internal copy of the given string. The host can immediately free that string after pushing it.

Push a string built from a buffer

The host can also push onto the stack a string value built from a string buffer. The process for pushing one of those strings is as follows:

1. Declare the buffer as a variable of type `luaL_Buffer`.
2. Initialize the buffer using `luaL_buffinit()` or `luaL_buffinitsize()`.
3. Fill the buffer using calls to `luaL_addch()`, `luaL_addlstring()`, and `luaL_addvalue()`, or by filling in the string returned by `luaL_buffinitsize()`.
4. Push the final string onto the stack using `luaL_pushresult()` or `luaL_pushresultsize()`.

CAUTION

While a buffer is in use, it utilizes a variable number of stack elements. Any non-buffer-related values that are pushed onto the stack should be popped prior to appending to the buffer.

Example 23 demonstrates how to push a string built from a string buffer whose final length is unknown ahead of time and Example 24 demonstrates how to push a string whose final length is known ahead of time.

Example 23. Push the entire contents of a file as a string

```
FILE *f = fopen(filename, "r");
luaL_Buffer b;
luaL_buffinit(L, &b);
char buf[BUFSIZ];
while (fgets(buf, BUFSIZ, f) != NULL)
  luaL_addlstring(&b, buf, strlen(buf));
luaL_pushresult(&b);
fclose(f);
```

Example 24. Push a lower-case copy of a string

```
luaL_Buffer b;
size_t len = strlen(s);
char *p = luaL_buffinitsize(L, &b, len);
for (int i = 0; i < len; i++)
  p[i] = tolower((unsigned char)s[i]);
luaL_pushresultsize(&b, len);
```

`luaL_Buffer`
: The C type associated with a Lua string buffer.

`void luaL_buffinit(lua_State *L, luaL_Buffer *b);`
: Initializes buffer *b*, a previously declared variable.

`char *luaL_buffinitsize(lua_State *L, luaL_Buffer *b, size_t len);` **Lua 5.2, 5.3**
: Initializes buffer *b*, a previously declared variable, and returns a string of length *len* bytes that can be filled in and subsequently added to *b* using `luaL_addsize()`.

`void luaL_addchar   (luaL_Buffer *b, char ch);`
`void luaL_addlstring(luaL_Buffer *b, const char *s, size_t len);`
: Adds to buffer *b* byte *ch* or string *s* of length *len* bytes.

`void luaL_addvalue(luaL_Buffer *b);`
: Pops a value off the stack and adds its string representation to buffer *b*.

`char *luaL_prepbuffer  (luaL_Buffer *b);`
`char *luaL_prepbuffsize(luaL_Buffer *b, size_t size);` **Lua 5.2, 5.3**
: Returns a string of length `LUAL_BUFFERSIZE` or *size* bytes that can be filled in and subsequently added to buffer *b* using `luaL_addsize()`.

`void luaL_addsize(luaL_Buffer *b, size_t n);`
: Adds to buffer *b* *n* bytes from the string returned by `luaL_buffinitsize()`, `luaL_prepbuffer()` or `luaL_prepbuffsize()`.

`void luaL_pushresult(luaL_Buffer *b);`
: Pushes onto the stack the value of buffer *b*.

`void luaL_pushresultsize(luaL_Buffer *b, size_t n);` **Lua 5.2, 5.3**
: Adds to buffer *b* *n* bytes from the string returned by `luaL_buffinitsize()`, `luaL_prepbuffer()` or `luaL_prepbuff`

`size()`, and pushes onto the stack the resulting value of *b*.

Push a function

The host can push C function values onto the stack. However, not just any arbitrary C function can be pushed, but only those of type `lua_CFunction` that follow Lua's convention. The section "C Functions" on page 108 describes C functions in more detail.

Just as Lua functions can have upvalues (non-local, non-global variables), C functions can have them too. Upvalues in C functions act just like C static variables and are available only in those functions. This is useful when functions need access to values that are neither arguments nor global variables. Example 28 on page 110 defines an upvalue to be the default value for a C function's table argument.

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

Pushes C function value *f* onto the stack.

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

Pops *n* values off the stack, associates them with C function *f* as upvalues, and pushes the resulting function (also called a *closure*) onto the stack.

f can use `lua_upvalueindex(i)` to fetch the stack index of upvalue number *i*, and through that index, retrieve the upvalue itself. The last value popped is the first upvalue (`i = 1`) and the first value popped is the last upvalue (`i = n`).

The maximum value for *n* is `256`.

Push a table

The host can push only empty table values onto the stack (and fill them in later), since there is no C type for tables.

```
void lua_newtable   (lua_State *L);
void lua_createtable(lua_State *L, int nlist, int nhash);
```

Creates and pushes onto the stack a new, empty table with *nlist* pre-allocated list elements and *nhash* pre-allocated hash values.

List elements have integer keys from `1` to *`nlist`*, and hash values have keys of any other valid value.

TIP

`lua_createtable()` exists purely for performance reasons when table size and makeup are known ahead of time. All tables automatically grow in size as needed.

Push a thread

The host can push thread values onto the stack. The section "Threading in C" on page 126 covers how to work with thread values.

`lua_State *lua_newthread(lua_State *L);`
: Creates and pushes onto the stack a new (suspended) thread and returns a pointer to it. The new thread has its own stack, but shares the same global environment as Lua interpreter *`L`*.

`int lua_pushthread(lua_State *thread);`
: Pushes onto the stack thread *`thread`* and returns `1` if *`thread`* is the main thread (i.e. it was created with `luaL_newstate()`).

Push a userdata

The host can push onto the stack an instance of a C data type (typically a C `struct`) as a *full userdata* value. The host can also push onto the stack a regular C pointer as a *light userdata* value. Userdata values are treated like any other Lua value. By assigning a full userdata a metatable, that value can act like an object. (A light userdata cannot have a metatable.)

When pushing a full userdata onto the stack, Lua allocates a raw block of memory for it. The host is free to fill in and manipulate that block of memory as it sees fit. Since Lua itself cannot modify userdata values, the host is assured of data integrity. When Lua detects a full userdata is no longer in use, it frees the memory associated with it.

When pushing a light userdata onto the stack, Lua does not assume any responsibility for managing that value. The host is still obligated to do so.

TIP

When a full userdata value is assigned a metatable with the metamethod `__gc()`, that metamethod will be called (with that userdata value as an argument) before Lua deletes the userdata. This allows the host to clean up anything outside of Lua related to that userdata, such as open files, extra host-allocated memory, etc. The section "Assign a Metatable" on page 117 describes how to assign a metatable to a value.

Example 25 demonstrates how a C `FILE*` pointer can be used as a Lua value. Example 30 on page 119 provides a more complete picture of userdata by using C99's complex data types as Lua objects in a complex number module.

Example 25. Use a C structure as a Lua value

```
// C struct for using FILE* as a Lua value.
typedef struct {
  FILE *f;
  int closed; // cannot fclose(f) twice
} lFile;

// Metamethod for closing files prior to deletion.
static int l_filegc(lua_State *L) {
  lFile *f = (lFile *)luaL_checkudata(L, 1, "file_mt");
  if (!f->closed)
    fclose(f->f);
  return 0;
}

/* ... */

// Create a new file userdata, open and associate a
// file with it, and assign a metatable that ensures
// the file is eventually closed.
lFile *f = (lFile *)lua_newuserdata(L, sizeof(lFile));
f->f = fopen(filename, "r");
f->closed = 0;
if (luaL_newmetatable(L, "file_mt")) {
  lua_pushcfunction(L, l_filegc);
  lua_setfield(L, -2, "__gc");
  /* define additional metamethods... */
}
```

```
lua_setmetatable(L, -2);
/* do something with the file... */
lua_pop(L, 1); // invokes __gc()
```

`void *lua_newuserdata(lua_State *L, size_t size);`
: Allocates *size* bytes of memory, pushes it onto the stack as a userdata value, and returns a pointer to the allocated memory.

`void lua_pushlightuserdata(lua_State *L, void *p);`
: Pushes light userdata value *p* onto the stack.

Push an arbitrary value

The host can push onto the stack another reference to a value already on the stack.

`void lua_pushvalue(lua_State *L, int index);`
: Pushes onto the stack another reference to the value at stack index *index*.

Pop Values

The host can explicitly pop values off the stack. Even though some Lua API functions pop certain values off the stack, the host should not rely on Lua to manage the stack properly. All values pushed must eventually be popped in order to prevent a stack overflow.

NOTE

Once a value is popped off the stack, if it has no more references to it (i.e. it is no longer in use), Lua will delete that value and free the memory associated with it. Temporarily storing values in Lua's registry table is one way to prevent this from happening. The section "Reference Operations" on page 107 describes Lua's reference system.

`void lua_pop(lua_State *L, int n);`
: Pops *n* values off the stack.

`void lua_remove(lua_State *L, int index);`
: Removes the value at stack index *index*, shifting stack values above it towards the bottom of the stack.

Query Values

The host can query the stack for what types of values are at particular stack indices.

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

Returns `1` if there is no value at stack index *`index`* or if that value is `nil`. Otherwise, returns `0`.

```
int lua_isnil           (lua_State *L, int index);
int lua_isboolean       (lua_State *L, int index);
int lua_isinteger       (lua_State *L, int index);      Lua 5.3
int lua_isnumber        (lua_State *L, int index);
int lua_isstring        (lua_State *L, int index);
int lua_istable         (lua_State *L, int index);
int lua_isfunction      (lua_State *L, int index);
int lua_iscfunction     (lua_State *L, int index);
int lua_isthread        (lua_State *L, int index);
int lua_isuserdata      (lua_State *L, int index);
int lua_islightuserdata(lua_State *L, int index);
```

Returns `1` if the value at stack index *`index`* is a nil, boolean, integer, number (either an integer or a float), string, table, function (either Lua or C), C function, thread, userdata (either full or light), or light userdata value. Otherwise, returns `0`.

`lua_isnumber()` and `lua_isstring()` will return `1` if the value is convertible to a number or string, respectively. `lua_type()` may be more applicable in those cases.

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

Returns the type of value at stack index *`index`*: `LUA_TNONE` for a non-existent value, `LUA_TNIL` for nil, `LUA_TBOOLEAN` for a boolean, `LUA_TNUMBER` for an integer or float, `LUA_TSTRING` for a string, `LUA_TTABLE` for a table, `LUA_TFUNCTION` for a function, `LUA_TTHREAD` for a thread, `LUA_TUSERDATA` for a userdata, or `LUA_TLIGHTUSERDATA` for a light userdata.

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

Returns the string name of value type *`type`*, which must be one of the values returned by `lua_type()`.

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

Returns the string name of the type of value at stack index *`index`*.

Retrieve Values

The host can retrieve Lua values that are on the stack, converted to C values. The means for doing so are broken up into sections that cover how to retrieve boolean, number, string, function, thread, and userdata values. (Nil and table values cannot be converted to C values.)

Retrieve a boolean

The host can retrieve boolean values that are on the stack, as well as other types of stack values converted to booleans.

`int lua_toboolean(lua_State *L, int index);`
: Returns the value at stack index *`index`* converted to a boolean, where any value other than `false` and `nil` is considered boolean true.

Retrieve a number

The host can retrieve number values that are on the stack. Lua provides some C type definitions that differentiate between integers and floats, since Lua numbers can be either.

`lua_Integer`
: The C type associated with Lua integers (typically `long long` in Lua 5.3 and `ptrdiff_t` in Lua 5.1 and 5.2). This is configurable when compiling Lua.

`lua_Unsigned` **Lua 5.2, 5.3**
: The C type associated with unsigned Lua integers (typically `unsigned long long` in Lua 5.3 and `unsigned long` in Lua 5.2). This is configurable when compiling Lua.

`lua_Number`
: The C type associated with Lua floats (typically `double`). This is configurable when compiling Lua.

```
lua_Integer  lua_tointeger  (lua_State *L, int index);
lua_Integer  lua_tointegerx (lua_State *L, int index,
                             int *isnum);          Lua 5.2, 5.3
lua_Unsigned lua_tounsigned (lua_State *L, int index); Lua 5.2
lua_Unsigned lua_tounsignedx(lua_State *L, int index,
                             int *isnum);               Lua 5.2
```

```
lua_Number   lua_tonumber   (lua_State *L, int index);
lua_Number   lua_tonumberx  (lua_State *L, int index,
                             int *isnum);
```
Lua 5.2, 5.3

Returns the value at stack index *index* converted to an integer or float, and sets *isnum* to `1` or, if the conversion fails, returns `0` and sets *isnum* to `0`.

Retrieve a string

The host can retrieve string values that are on the stack.

> **CAUTION**
>
> The string pointers returned by Lua are guaranteed to be valid only for as long as the value remains on the stack.

```
const char *lua_tostring  (lua_State *L, int index);
const char *lua_tolstring (lua_State *L, int index,
                           size_t *len);
const char *luaL_tolstring(lua_State *L, int index,
                           size_t *len);
```
Lua 5.2, 5.3

Returns the value at stack index *index* converted to a C-style string and sets *len* to the byte length of the returned string or, if the conversion fails, returns `NULL`.

If the value has the metamethod `__tostring()`, `luaL_tolstring()` calls that metamethod and returns the resulting string value instead. The section "Function Metamethods" on page 36 covers this metamethod in its generic form.

> **CAUTION**
>
> Calling any of these functions on a number value will actually change that number value into a string in the process. This may have undesirable side-effects, most notably during table iteration with numeric keys.

Retrieve a function

The host can retrieve C function values (not Lua function values) that are on the stack.

`lua_CFunction lua_tocfunction(lua_State *L, int index);`
: Returns the value at stack index *index* converted to a C function or, if the conversion fails, returns `NULL`.

Retrieve a thread

The host can retrieve thread values that are on the stack.

`lua_State *lua_tothread(lua_State *L, int index);`
: Returns the value at stack index *index* converted to a thread or, if the conversion fails, returns `NULL`.

Retrieve a userdata

The host can retrieve userdata values that are on the stack, regardless of whether they are full userdata or light userdata.

`void *lua_touserdata(lua_State *L, int index);`
: Returns the value at stack index *index* converted to a userdata or, if the conversion fails, returns `NULL`.

Retrieve an arbitrary value

The host can retrieve the raw C pointer for a table, function, thread, or userdata value that is on the stack. However, this pointer has little practical use and is guaranteed to be valid only for as long as that value remains on the stack.

`const void *lua_topointer(lua_State *L, int index);`
: Returns the table, function, thread, or userdata value at stack index *index* converted to a raw C pointer.

  This is typically used only for hashing or debugging, as there is no way to retrieve the Lua value associated with a raw pointer.

Basic Stack Operations

The host can perform many different operations on stack values, such as element, global variable, arithmetic, relational, bitwise, string, length, and reference operations. The following sections cover these operations.

Element Operations

The host can perform simple stack element operations.

`void lua_copy(lua_State *L, int from, int to);` **Lua 5.2, 5.3**
: Copies the value at stack index *from* to stack index *to*, overwriting the existing value.

`void lua_insert(lua_State *L, int index);`
: Moves the value at the top of the stack to stack index *index*, shifting prior stack values towards the top of the stack.

`void lua_replace(lua_State *L, int index);`
: Pops a value off the stack and moves it to stack index *index*, overwriting the existing value.

`void lua_rotate(lua_State *L, int index, int n);` **Lua 5.3**
: Rotates the stack values between stack index *index* and the top of the stack (inclusive) by *n* positions towards the top of the stack. *n* can be negative.

Global Variable Operations

The host can define and retrieve global Lua variables.

`void lua_setglobal(lua_State *L, const char *name);`
: Pops a value off the stack and assigns it to the global variable whose name is string *name*.

`int  lua_getglobal(lua_State *L, const char *name);` **Lua 5.3**
`void lua_getglobal(lua_State *L, const char *name);` **Lua 5.1, 5.2**
: Pushes onto the stack the value associated with the global variable whose name is string *name*, and returns the pushed value's type.

Arithmetic Operations

The host can invoke Lua's arithmetic operators. These operators may in turn invoke arithmetic metamethods, which are described in the section "Arithmetic Metamethods" on page 32.

```
void lua_arith(lua_State *L, LUA_OPADD);               Lua 5.2, 5.3
void lua_arith(lua_State *L, LUA_OPSUB);               Lua 5.2, 5.3
void lua_arith(lua_State *L, LUA_OPMUL);               Lua 5.2, 5.3
void lua_arith(lua_State *L, LUA_OPDIV);               Lua 5.2, 5.3
void lua_arith(lua_State *L, LUA_OPIDIV);                   Lua 5.3
void lua_arith(lua_State *L, LUA_OPMOD);               Lua 5.2, 5.3
void lua_arith(lua_State *L, LUA_OPPOW);               Lua 5.2, 5.3
```

Pops two values off the stack, adds (+), subtracts (-), multiplies (*), divides (/), integer divides (//), computes the remainder of floor division between (%), or exponentiates (^) them, and pushes the resulting value onto the stack.

The first operand is the second value popped, and the second operand is the first value popped.

```
void lua_arith(lua_State *L, LUA_OPUNM);               Lua 5.2, 5.3
```

Pops a value off the stack, negates (-) it, and pushes the resulting value onto the stack.

Relational Operations

The host can invoke Lua's relational operators. These operators may in turn invoke relational metamethods, which are described in the section "Relational Metamethods" on page 33.

```
int lua_compare (lua_State *L, int index1, int index2,
                 LUA_OPEQ);                            Lua 5.2, 5.3
int lua_equal   (lua_State *L, int index1,
                 int index2);                               Lua 5.1
int lua_compare (lua_State *L, int index1, int index2,
                 LUA_OPLT);                            Lua 5.2, 5.3
int lua_lessthan(lua_State *L, int index1,
                 int index2);                               Lua 5.1
int lua_compare (lua_State *L, int index1, int index2,
                 LUA_OPLE);                            Lua 5.2, 5.3
```

Compares the values at stack indices *index1* and *index2* for equality (==), less than (<), or less than or equal to (<=), and returns `1` if the comparison is correct or `0` if the comparison is incorrect.

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

Returns `1` if the values at stack indices *index1* and *index2* are equal, bypassing metamethods. Otherwise, returns `0`.

Bitwise Operations

The host can invoke Lua 5.3's bitwise operators. These operators may in turn invoke bitwise metamethods, which are described in the section "Bitwise Metamethods" on page 34.

`void lua_arith(lua_State *L, LUA_OPBAND);` **Lua 5.3**
`void lua_arith(lua_State *L, LUA_OPBOR);` **Lua 5.3**
`void lua_arith(lua_State *L, LUA_OPBXOR);` **Lua 5.3**

Pops two values off the stack, performs bitwise AND (&), OR (|), or XOR (~) on them, and pushes the resulting value onto the stack.

The first operand is the second value popped, and the second operand is the first value popped.

`void lua_arith(lua_State *L, LUA_OPBNOT);` **Lua 5.3**

Pops a value off the stack, performs bitwise NOT (~) on it, and pushes the resulting value onto the stack.

`void lua_arith(lua_State *L, LUA_OPSHL);` **Lua 5.3**
`void lua_arith(lua_State *L, LUA_OPSHR);` **Lua 5.3**

Pops two values off the stack, performs left shift (<<) or right shift (>>) on them, and pushes the resulting value onto the stack.

The first operand is the second value popped, and the second operand is the first value popped.

String Operations

The host can invoke Lua's string concatenation (..) and length (#) operators. These operators may in turn invoke their respective metamethods, which are covered in the section "Other Operator and Statement Metamethods" on page 34. The host can also take advantage of a convenience function for performing global substitution in strings.

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

Pops *n* values off the stack, concatenates them as strings, and pushes the resulting value onto the stack.

`void lua_len(lua_State *L, int index);` **Lua 5.2, 5.3**

Pushes onto the stack the length (#) of the value at stack index *index*.

```
lua_Integer luaL_len  (lua_State *L, int index);      Lua 5.3
int         luaL_len  (lua_State *L, int index);      Lua 5.2
size_t      lua_objlen(lua_State *L, int index);      Lua 5.1
```

Returns the length (#) of the value at stack index *index*.

```
const char *luaL_gsub(lua_State *L, const char *s,
                      const char *sub, const char *repl);
```

Pushes onto the stack a copy of string *s* with all instances of substring *sub* replaced with string *repl*, and returns the new string.

sub is not interpreted as a Lua pattern.

Table Operations

The host can interact with tables in many different ways. The following sections describe how to retrieve the value associated with a table key, how to assign a value to a table key, and how to iterate over a table's key-value pairs.

Retrieve the value assigned to a key

The host can retrieve values assigned to table keys. These operations may invoke the metafield `__index` or metamethod `__index()`, both of which are covered in the section "Other Operator and Statement Metamethods" on page 34.

```
int  lua_gettable(lua_State *L, int index);           Lua 5.3
void lua_gettable(lua_State *L, int index);      Lua 5.1, 5.2
int  lua_rawget  (lua_State *L, int index);           Lua 5.3
void lua_rawget  (lua_State *L, int index);      Lua 5.1, 5.2
```

Pushes onto the stack the value in the table at stack index *index* associated with the key at the top of the stack, and returns the pushed value's type.

`lua_rawget()` bypasses all metamethods.

```
int  lua_geti   (lua_State *L, int index,
                 lua_Integer i);                      Lua 5.3
int  lua_rawgeti(lua_State *L, int index,
                 lua_Integer i);                      Lua 5.3
void lua_rawgeti(lua_State *L, int index, int i);  Lua 5.1, 5.2
```

Pushes onto the stack the *i*-th element of the list at stack index *index*, and returns the pushed element's type.

`lua_rawgeti()` bypasses all metamethods.

```
int  lua_getfield(lua_State *L, int index,
                  const char *key);
```
Lua 5.3

```
void lua_getfield(lua_State *L, int index,
                  const char *key);
```
Lua 5.1, 5.2

Pushes onto the stack the value associated with string *`key`* in the table at stack index *`index`*, and returns the pushed value's type.

```
int  lua_rawgetp(lua_State *L, int index,
                 const void *p);
```
Lua 5.3

```
void lua_rawgetp(lua_State *L, int index,
                 const void *p);
```
Lua 5.2

Pushes onto the stack the value associated with light userdata *`p`* in the table at stack index *`index`* (bypassing all metamethods), and returns the pushed value's type.

```
int luaL_getsubtable(lua_State *L, int index,
                     const char *key);
```
Lua 5.2, 5.3

Pushes onto the stack an existing or newly created table value assigned to string *`key`* in the table at stack index *`index`*, and returns `1` if the pushed table already existed or `0` if it was created.

Assign a value to a key

The host can fill in a table with key-value pairs. These operations may invoke the `__newindex` metafield or `__newindex()` metamethod, both of which are covered in the section "Other Operator and Statement Metamethods" on page 34.

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

Pops two values off the stack and associates them as a key-value pair in the table at stack index *`index`*.

The second value popped is the key and the first value popped is the value.

`lua_rawset()` bypasses all metamethods.

```
void lua_seti   (lua_State *L, int index,
                 lua_Integer i);                    Lua 5.3
void lua_rawseti(lua_State *L, int index,
                 lua_Integer i);                    Lua 5.3
void lua_rawseti(lua_State *L, int index, int i);   Lua 5.1, 5.2
```

Pops a value off the stack and makes it the *i*-th element in the list at stack index *index*.

`lua_rawseti()` bypasses all metamethods.

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

Pops a value off the stack and associates it with string *key* to make a key-value pair in the table at stack index *index*.

```
void lua_rawsetp(lua_State *L, int index,
                 const void *p);                    Lua 5.2, 5.3
```

Pops a value off the stack and associates it with light userdata *p* to make a key-value pair in the table at stack index *index*, bypassing all metamethods.

TIP

Light userdata can be used as unique keys in Lua's registry table without having to appeal to Lua's reference system. The section "Reference Operations" on page 107 describes the registry.

Iterate over a table

The host can iterate over all key-value pairs in a table using the following procedure:

1. Push the table to be iterated over onto the stack.
2. Push the value `nil` using `lua_pushnil()`.
3. Continually call `lua_next()` while its return value is nonzero.
4. For each iteration, a key is just below the top of the stack and its associated value is at the top of the stack.
5. Before the next iteration, pop the value off the stack, leaving the current key.
6. If a new key-value pair was added during the iteration, pop the key as well and push `nil` in order to restart iter-

ation from the beginning. (Any key-value pairs edited or deleted during iteration do not require a restart.)

Iteration order is not defined, even if the table is a list.

CAUTION

If a key is numeric, calling `lua_tostring()` or `lua_tolstring()` on it will actually change that key into a string and adversely affect the next call to `lua_next()`.

Example 26 illustrates how to iterate over a table and delete all of its key-value pairs whose keys are strings.

Example 26. Delete all string keys from a table

```
/* push table to be iterated over... */
lua_pushnil(L);
while (lua_next(L, -2) != 0) {
  if (lua_type(L, -2) == LUA_TSTRING) {
    // Delete values assigned to string keys (fields).
    const char *key = lua_tostring(L, -2);
    lua_pushnil(L);
    lua_setfield(L, -4, key);
  }
  lua_pop(L, 1); // value
}
lua_pop(L, 1); // table iterated over
```

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

Pops a key off the stack and pushes onto the stack the next key-value pair from the table at stack index *`index`*. If there are no more key-value pairs to push, returns `0`.

The pushed value is at the top of the stack and the pushed key is just below it.

NOTE

Modifying a table during traversal is permitted as long as no new key-value pairs are added. If a new pair is added, traversal must begin anew.

Length Operations

The host can invoke Lua's length operator. This operator may in turn invoke the length metamethod, which is covered in the section "Other Operator and Statement Metamethods" on page 34.

```
void lua_len(lua_State *L, int index);                Lua 5.2, 5.3
```

Pushes onto the stack the length (#) of the value at stack index *index*.

```
lua_Integer luaL_len(lua_State *L, int index);        Lua 5.3
int         luaL_len(lua_State *L, int index);        Lua 5.2
```

Returns the length (#) of the value at stack index *index*.

```
size_t lua_rawlen(lua_State *L, int index);           Lua 5.2, 5.3
size_t lua_objlen(lua_State *L, int index);           Lua 5.1
```

Returns the length of the string, table, or userdata value at stack index *index*, bypassing all metamethods.

Reference Operations

The stack is only meant for storing temporary values prior to performing a stack operation. (Once a value is popped from the stack, Lua may garbage collect it.) When the host needs to store values for later use, it can either assign those values to global Lua variables (which may not be ideal), or use an internal *registry* table that Lua provides for storing and retrieving any Lua values. Lua's registry exists at the special stack index `LUA_REGISTRYINDEX` (which is not a true stack index, so it cannot be popped, removed, replaced, rotated, etc.). The registry is accessible only through Lua's C API, ensuring integrity.[10]

NOTE

By convention, string keys comprising an underscore followed by one or more upper-case letters are reserved for use by Lua itself in its registry.

Since the registry is also available to any external Lua C modules the host loads, there is a possibility of key clashes. In or-

10 Technically, Lua's standard library module `debug` can access the registry, but the host can choose not to load that module or to disable it.

der to avoid this, Lua provides a way to store and retrieve unique references to Lua values in the registry (but does not require the host to utilize it). Example 31 on page 124 uses the registry to store and retrieve a sandboxed environment for running potentially unsafe code in.

CAUTION

When manually adding key-value pairs to Lua's registry, integer keys may not be used, as that will interfere with Lua's unique reference system.

`int luaL_ref(lua_State *L, LUA_REGISTRYINDEX);`
: Pops a value off the stack, creates a unique integer reference to it in Lua's registry table, and returns that reference.

  The referenced value will not be eligible for garbage collection at least until `luaL_unref()` is called for that value.

`int lua_rawgeti(lua_State *L, LUA_REGISTRYINDEX, int ref);`
: Pushes onto the stack the value associated with the unique integer reference *`ref`* returned by `luaL_ref()`, and returns the pushed value's type.

`void luaL_unref(lua_State *L, LUA_REGISTRYINDEX, int ref);`
: Releases integer reference *`ref`* to the value in Lua's registry table. That value may now be garbage collected if it is no longer being used.

C Functions

A C function is a special kind of function that Lua can interact with. It is just like a normal C function, except it has a specific type:

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

Functions of this type receive their arguments from the stack and push their return values onto the stack. C functions are a subset of Lua's first-class function values and behave in exactly the same way. The following sections describe how to define, register, and call C functions.

Define a C Function

C functions are defined using the type `lua_CFunction` and follow the form of a normal C function definition. When a C function is called, it receives its own stack, which contains only the argument values passed to that function (the first argument is at the bottom of the stack and the last argument is at the top of the stack). When the C function is finished, it should push its return values onto its stack (starting with the first return value) and then return the number of return values pushed. Example 27 defines and makes available a simple C function that returns the value of C99's gamma function for a given number argument.

Example 27. Mathematical gamma function

```
static int l_gamma(lua_State *L) {
  double z = luaL_checknumber(L, 1); // fetch argument
  lua_pushnumber(L, tgamma(z)); // push value to return
  return 1; // number of stack values to return
}

/* ... */

// Add gamma to Lua's math module.
lua_getglobal(L, "math");
lua_pushcfunction(L, l_gamma);
lua_setfield(L, -2, "gamma");
lua_pop(L, 1); // global "math"
```

A C function's stack is independent of the "main" stack and any other active C function stack. The function is not required to pop argument values off its stack, as the stack is discarded after the function returns. (The function is not even required to pop off any intermediate values it pushed, so long as there is enough stack space for its return values.)

Lua provides a number of convenient API functions designed specifically for C functions. These functions are broken up into sections that cover how to validate and retrieve argument values, how to retrieve upvalues, and how to prevent a stack overflow. (C functions are not limited to using these API functions, however.) Example 28 exhibits a few of these convenience functions and concepts.

Example 28. C function that translates string characters

```
static int translate_chars(lua_State *L) {
  // Fetch arguments. The first should be a string. The
  // second should be a table, if given. Otherwise, use
  // a default table stored as an upvalue.
  const char *s = luaL_checkstring(L, 1);
  if (lua_gettop(L) > 1)
    luaL_checktype(L, 2, LUA_TTABLE);
  else
    lua_pushvalue(L, lua_upvalueindex(1));

  // Allocate and fill a copy of the string argument,
  // translate its characters according to the table
  // argument, and push the result.
  char *o = strcpy(malloc(strlen(s) + 1), s);
  for (char *p = o; *p; p++) {
    lua_pushlstring(L, p, 1); // table key
    lua_gettable(L, 2); // fetch value assigned to key
    if (lua_isstring(L, -1))
      *p = *lua_tostring(L, -1); // translate char
    lua_pop(L, 2); // table key and value
  }
  lua_pushstring(L, o); // push the value to return
  free(o);
  return 1; // the number of stack values to return
}

/* ... */

// Create the default translation table, assign it as
// an upvalue to translate_chars, and register that
// function as the global function "tr".
lua_createtable(L, 0, 1);
lua_pushliteral(L, "_");
lua_setfield(L, -2, " "); // translate ' ' to '_'
lua_pushcclosure(L, translate_chars, 1);
lua_setglobal(L, "tr");

-- Lua code.
tr("hello world!") -- returns "hello_world!"
tr("hello!", {["!"] = "?"}) -- returns "hello?"
```

Validate and retrieve argument value types

The host can conveniently validate argument value types while retrieving them converted to C values.

```
lua_Integer  luaL_checkinteger (lua_State *L, int arg);
int          luaL_checkint     (lua_State *L,
                                int arg);          Lua 5.1, 5.2
long         luaL_checklong    (lua_State *L,
                                int arg);          Lua 5.1, 5.2
lua_Unsigned luaL_checkunsigned(lua_State *L,
                                int arg);              Lua 5.2
lua_Number   luaL_checknumber  (lua_State *L, int arg);
const char  *luaL_checkstring  (lua_State *L, int arg);
void        *luaL_checkudata   (lua_State *L, int arg,
                                const char *name);
```

Asserts that function argument number *arg* is an integer value, number value (either an integer or a float), string value, or userdata value whose metatable is the metatable identified by string *name*, and returns that value converted to its respective C type, or raises an error.

```
void *luaL_testudata(lua_State *L, int arg,
                     const char *name);          Lua 5.2, 5.3
```

Returns the value of function argument number *arg* converted to a userdata, provided its metatable is the metatable identified by string *name*, or returns `NULL` if that value is not the desired type of userdata.

```
void luaL_checktype(lua_State *L, int arg, int type);
```

Asserts that function argument number *arg* is Lua type *type*, or raises an error.

```
int luaL_typerror(lua_State *L, int arg,
                  const char *name);                 Lua 5.1
```

Raises an error that function argument number *arg* is not of string type *name*. *name* is typically the name of a custom userdata type.

Validate argument values

The host can conveniently validate that argument values exist or satisfy a condition.

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

Asserts that function argument number *arg* was given, or raises an error.

```
void luaL_argcheck(lua_State *L, int expr, int arg,
                   const char *message);
```

Asserts that expression *expr* evaluates to a non-zero value, or raises an error that implicates function argument number *arg* with string *message* as additional error information.

```
void luaL_argerror(lua_State *L, int arg,
                   const char *message);
```

Raises an error that implicates function argument number *arg* with string *message* as additional error information.

Specify default argument values

The host can conveniently retrieve argument values converted to C values, or retrieve default values.

```
lua_Integer  luaL_optinteger (lua_State *L, int arg,
                              lua_Integer default);
int          luaL_optint     (lua_State *L, int arg,
                              int default);          Lua 5.1, 5.2
long         luaL_optlong    (lua_State *L, int arg,
                              long default);         Lua 5.1, 5.2
lua_Unsigned luaL_optunsigned(lua_State *L, int arg,
                              lua_Unsigned default);   Lua 5.2
lua_Number   luaL_optnumber  (lua_State *L, int arg,
                              lua_Number default);
const char  *luaL_optstring  (lua_State *L, int arg,
                              const char *default);
const char  *luaL_optlstring (lua_State *L, int arg,
                              const char *default,
                              size_t *len);
```

Returns the value of function argument number *arg* converted to an integer, float, or string, defaulting to *default* if the argument value does not exist or is `nil`, or, if the conversion fails, raises an error.

`luaL_optlstring()` sets *len* to the byte length of the returned string.

```
int luaL_checkoption(lua_State *L, int arg,
                     const char *default,
                     const char *const list[]);
```

Asserts that function argument number *arg* is a string included in `NULL`-terminated string list *list*, and returns the index of that string in *list*, or raises an error. If given, the default value for argument number *arg* is string *default*.

Retrieve upvalue indices

The host can retrieve the stack indices of a C function's upvalues.

`int lua_upvalueindex(int i);`
: Returns the stack index of the *i*-th upvalue of the current function.

  The returned index can be used in most API functions involving stack indices, but since it is not a true stack index, it cannot be popped, removed, replaced, rotated, etc.

Raise an error

The host can raise errors from within C functions. Raising an error outside of a C function triggers Lua's panic function and will most likely result in a hard abort.

`int luaL_error(lua_State *L, const char *format, ...);`
: Raises an error with a formatted error message constructed from string *format* and a variable number of arguments. *format* contains a sequence of placeholders that specify how to format their respective arguments. Table 14 on page 89 lists valid placeholders along with their meanings.

  If available, filename and line number information is automatically prepended to the error message.

`int lua_error(lua_State *L);`
: Raises a Lua error whose error message is at the top of the stack.

> **TIP**
>
> The statements "`return luaL_error(L, ...);`" and "`return lua_error(L);`" are idioms in C functions, signaling that the function immediately halts execution.

Increase stack size

A C function's initial stack size is n + 20 elements, where n is the number of argument values already on the stack when the function is called. (This default size is configurable when compiling Lua.) The stack does not grow automatically as values are pushed onto it, so the host needs to grow it as necessary prior to pushing values in order to prevent a stack overflow.

`void luaL_checkstack(lua_State *L, int n, const char *msg);`
: Asserts that the stack can grow by *n* more values, or raises an error with error message string *msg*.

> **NOTE**
>
> The maximum stack size is 8,000 elements in Lua 5.1 and 15,000 elements in Lua 5.2 and 5.3. This arbitrary limit is configurable when compiling Lua. It is possible to run out of memory before hitting the maximum stack size, especially in embedded environments.

Register a C Function

The host can conveniently assign a C function to a global variable. (C functions may also be assigned to table keys using various other API functions.)

```
void lua_register(lua_State *L, const char *name,
                  lua_CFunction f);
```
: Assigns C function *f* to the global variable whose name is string *name*.

Call a C Function

The host can call a C function (or any Lua function for that matter) using the following procedure:

1. Push the function to call onto the stack.
2. Push onto the stack the argument values to pass to the function, starting with the first argument value.
3. Call the function using one of Lua's API functions.
4. Process any resulting values returned by the function and pop them off the stack. (The last value returned is at the top of the stack.)

Example 29 demonstrates how to call the Lua function `str` ing.find() and handle the variable number of values it returns (zero in the case of no match, two in the case of a match with no captures, and three or more in the case of a match with captures).

Example 29. Call Lua's `string.find`

```
// Record initial stack size due to LUA_MULTRET.
int n = lua_gettop(L);
// Push the global function string.find().
lua_getglobal(L, "string");
lua_getfield(L, -1, "find");
lua_replace(L, -2);
// Push two arguments.
lua_pushstring(L, s);
lua_pushstring(L, pattern);
// Call the function with those two arguments,
// expecting a variable number of results.
if (lua_pcall(L, 2, LUA_MULTRET, 0) == LUA_OK† &&
    lua_gettop(L) > n) {
  int start = lua_tointeger(L, n + 1);
  int end = lua_tointeger(L, n + 2);
  /* process returned positions and any captures... */
  lua_settop(L, n); // pop all returned values
}
```

```
void lua_call  (lua_State *L, int nargs, int nresults);
int  lua_pcall (lua_State *L, int nargs, int nresults,
                int error_handler);
```
Lua 5.2

Pops *`nargs`* function argument values off the stack, pops off the stack the function that is now at the top of the stack, calls that popped function with the popped arguments (the last value popped being the first argument and the first value popped being the last argument), and pushes the first *`nresults`* values returned by the function onto the stack (or all of them if *`nresults`* is `LUA_MULTRET`). `lua_pcall()` returns `LUA_OK` (or `0` in Lua 5.1) on success.

`lua_call()` should only be called from within C functions that do not care to handle errors and have been ultimately invoked by a protected call. The section "Error Handling" on page 122 describes protected calls.

If an error occurs, `lua_pcall()` pushes the error message onto the stack and returns a non-zero error code. If *`er`*

† `LUA_OK` exists only in Lua 5.2 and 5.3. Lua 5.1 uses the constant 0 instead.

ror_handler is nonzero, the function at stack index *error_handler* is called with the error message as an argument, and that function's return value is the error message ultimately pushed onto the stack. Table 15 on page 123 lists Lua's error codes and their meanings.

If the value being called is a table or userdata value with the metamethod `__call()`, that metamethod is called to perform the operation. The section "Other Operator and Statement Metamethods" on page 34 covers this metamethod in its generic form.

`int lua_cpcall(lua_State *L, lua_CFunction f, void *userdata);` **Lua 5.1**

Calls C function *f* with userdata *userdata* as that function's only argument value and returns `0` on success.

If an error occurs, the error message is pushed onto the stack and a non-zero error code is returned instead. Table 15 on page 123 lists Lua's error codes and their meanings.

Metatables

The host can create metatables, assign and retrieve the metatables of values, call specific metamethods, and retrieve specific metafields. The means for doing so are described in the following sections. The section "Metatables and Metamethods" on page 31 describes metatables, metamethods, and metafields.

Create or Fetch a Metatable

The host can specifically create a metatable, as opposed to creating a generic table and using it as a metatable. The host can also easily fetch a previously created metatable by name.

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

Pushes onto the stack the metatable identified by string *name*, and returns `1` if the metatable had to be created first or `0` if the metatable already existed.

CAUTION

Lua keeps track of all metatable names in the same place. If the host loads any external C modules, those modules will also have the ability to create their own metatables, so there is a possibility of name clashes.

Assign a Metatable

The host can assign a metatable to a value (bypassing the metafield `__metatable` that value may have). In the C API, values are not limited to tables and userdata, but can be any Lua value. However, only tables and userdata can have individual metatables. All other types each share a single metatable.

```
void lua_setmetatable(lua_State *L, int index);      Lua 5.2, 5.3
int  lua_setmetatable(lua_State *L, int index);      Lua 5.1
```

Pops a table value off the stack and assigns it to be the metatable of the value at stack index *`index`*. Always returns `1` in Lua 5.1.

```
void luaL_setmetatable(lua_State *L,
                       const char *name);            Lua 5.2, 5.3
```

Assigns the metatable identified by string *`name`* to be the metatable of the value at the top of the stack.

Retrieve a Metatable

The host can retrieve a value's metatable.

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

Pushes onto the stack the metatable associated with the value at stack index *`index`* and returns `1`, or, if that value has no metatable, pushes nothing and returns `0`.

```
int  luaL_getmetatable(lua_State *L,
                       const char *name);            Lua 5.3
void luaL_getmetatable(lua_State *L,
                       const char *name);            Lua 5.1, 5.2
```

Pushes onto the stack the metatable identified by string *`name`* or `nil` if no metatable was found, and returns the pushed value's type.

Metamethods and Metafields

The host can call specific metamethods and retrieve specific metafields.

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

Calls the metamethod named string *name* that belongs to the metatable associated with the value at stack index *index*, pushes onto the stack the value returned by that call, and returns `1`. If the metamethod does not exist, returns `0` and pushes nothing. The metamethod is passed the stack value as its only argument.

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

Pushes onto the stack the value associated with string *key* in the metatable associated with the value at stack index *index*, and returns the pushed value's type. If the metafield does not exist, returns `LUA_TNIL` and pushes nothing.

C Modules

Lua provides an API for creating loadable C modules, which are typically just Lua tables. A C module often contains:

- A set of Lua C functions specific to the module.
- An array of type `luaL_Reg[]` that maps those C functions to string names in the module's table.
- A Lua C function that serves as the module's entry point. This function creates the module table and pushes it onto the stack as a return value. By convention, the function's name is "`luaopen_`*name*," where *name* is the module's actual name (the string that would be passed to Lua's `require()` function). Any '`.`' characters in *name* should be replaced with '`_`' and any "`-`*version*" suffix should be ignored. For example, a module named "lpeg" has the entry point "`luaopen_lpeg`", a submodule named "utf8.ext" has the entry point "`luaopen_utf8_ext`", and a versioned submodule named "utf8.ext-v2" has the same entry point "`luaopen_utf8_ext`".

Example 30 lists a module that provides an interface to C99's complex numbers.

Example 30. Complex number module

```
#include <complex.h>
#include "lua.h"
#include "lauxlib.h"
typedef double complex Complex;

// Pushes a complex number as userdata.
static int l_pushcomplex(lua_State *L, Complex z) {
  Complex *p = lua_newuserdata(L, sizeof(Complex));
  *p = z;
  luaL_setmetatable(L, "complex_mt");
  return 1;
}

// Creates and pushes a new complex number.
static int l_cnew(lua_State *L) {
  double x = luaL_optnumber(L, 1, 0);
  double y = luaL_optnumber(L, 2, 0);
  l_pushcomplex(L, x + y * I);
  return 1;
}

// Asserts and returns a complex number function
// argument.
static Complex lL_checkcomplex(lua_State *L, int arg) {
  if (lua_isuserdata(L, 1))
    return *((Complex *)luaL_checkudata(L, arg,
                                        "complex_mt"));
  else
    return luaL_checknumber(L, arg);
}

// Defines a unary complex number operation.
#define unop(name, op) \
  static int l_c##name(lua_State *L) { \
    Complex z = lL_checkcomplex(L, 1); \
    return l_pushcomplex(L, op(z)); \
  }

// Defines a binary complex number operation.
#define binop(name, op) \
  static int l_c##name(lua_State *L) { \
    Complex z1 = lL_checkcomplex(L, 1); \
```

```
    Complex z2 = lL_checkcomplex(L, 2); \
    return l_pushcomplex(L, z1 op z2); \
  }

// Complex number operations.
unop(abs, cabs)
unop(real, creal)
unop(imag, cimag)
unop(arg, carg)
unop(conj, conj)
binop(add, +)
binop(sub, -)
binop(mul, *)
binop(div, /)
unop(unm, -)
binop(eq, ==)

// String representation of a complex number.
static int l_ctostring(lua_State *L) {
  Complex z = lL_checkcomplex(L, 1);
  double x = creal(z), y = cimag(z);
  if (x != 0 && y > 0)
    lua_pushfstring(L, "%f+%fi", x, y);
  else if (x != 0 && y < 0)
    lua_pushfstring(L, "%f%fi", x, y);
  else if (x == 0)
    lua_pushfstring(L, "%fi", y);
  else
    lua_pushfstring(L, "%f", x);
  return 1;
}

// Complex module functions.
static const luaL_Reg complex_functions[] = {
  {"new", l_cnew},
  {"abs", l_cabs},
  {"real", l_creal},
  {"imag", l_cimag},
  {"arg", l_carg},
  {"conj", l_cconj},
  {NULL, NULL}
};

// Complex number metamethods.
static const luaL_Reg complex_metamethods[] = {
  {"__add", l_cadd},
  {"__sub", l_csub},
```

```
  {"__mul", l_cmul},
  {"__div", l_cdiv},
  {"__unm", l_cunm},
  {"__eq", l_ceq},
  {"__tostring", l_ctostring},
  {NULL, NULL}
};

// Complex number module entry point.
int luaopen_complex(lua_State *L) {
  // Create and push the module table.
  luaL_newlib(L, complex_functions);
  // Create the complex number metatable, fill it,
  // link it with the module table, then pop it.
  luaL_newmetatable(L, "complex_mt");
  luaL_setfuncs(L, complex_metamethods, 0);
  lua_pushvalue(L, -2); // the module table
  lua_setfield(L, -2, "__index");
  lua_pop(L, 1); // metatable
  return 1; // return the module table
}

-- Lua code.
local complex = require("complex")
complex.new(3, 4) + complex(-1, -2) -- results in 2+2i
complex.new(-1, 1):conj() -- results in -1-1i
```

`luaL_Reg`

A C `struct` that represents a named C function:

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

`void luaL_newlib(lua_State *L,`
`              const luaL_Reg list[]);` **Lua 5.2, 5.3**

Pushes onto the stack a new table composed of the C functions in `NULL`-terminated list *`list`*.

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

Pushes onto the stack the metatable identified by string *`name`*, and returns `1` if the metatable had to be created first or `0` if the metatable already existed.

> **CAUTION**
>
> Lua keeps track of all metatable names in the same place. If the host loads any external C modules, those modules will also have the ability to create their own metatables, so there is a possibility of name clashes.

```
void luaL_setfuncs(lua_State *L, const luaL_Reg *list,
                   int n);
```
Lua 5.2, 5.3

Pops *n* values off the stack, associates them with the C functions in `NULL`-terminated list *list* as upvalues, and adds the resulting closures to the table that is now at the top of the stack, that was originally below the *n* values.

```
void luaL_requiref(lua_State *L, const char *name,
                   lua_CFunction f, int global);
```
Lua 5.2, 5.3

Mimics Lua's `require()` function by calling function *f* with string *name* as an argument and registering the value returned by *f* as the module named *name*. If *global* is nonzero, assigns the returned value to the global variable whose name is *name*. Only the first value returned by *f* is used and left on the stack.

Subsequent calls to `luaL_requiref()` with *name* will produce the original value returned by *f*.

```
void luaL_register(lua_State *L, const char *name,
                   const luaL_Reg *list);
```
Lua 5.1

Pushes onto the stack a new table composed of the C functions in `NULL`-terminated list *list*, registers that table as the module named *name*, and assigns it to the global variable whose name is *name*. If *name* is `NULL`, adds all functions in *list* to the table at the top of the stack.

Error Handling

Properly handling Lua errors in C is vitally important. Whenever Lua raises an error (either on its own or from an explicit API call), it uses C's function `longjmp()` in an attempt to handle the error. Unless the error occurred within a protected call, Lua's panic function is invoked, and a hard abort will occur unless the host intervenes and performs a `longjmp()` of its own to recover. By contrast, a protected call catches and handles the error gracefully, and returns an error code. The API functions lua_pcall(), lua_cpcall(), luaL_dofile(), luaL_dostring(), lua_resume(), and lua_pcallk() are all protected calls. The first two are described in the section "Call a C Function" on page 114, the next two are described in the section "Load and Run Dynamic Code" on page 124, and the last two are covered in the section "Threading in C" on page 126. Each of those sections has an example that demonstrates how to handle errors with their respective API functions. Table 15 lists the error codes that protected calls can return, along with their meanings.

Table 15. Error codes returned by protected calls

Error code	Meaning
LUA_OK[a]	Success
LUA_ERRRUN	Runtime error
LUA_ERRMEM	Memory allocation error
LUA_ERRERR	Error running the error handler given to lua_pcall() or lua_pcallk()

[a] Only in Lua 5.2 and 5.3. Lua 5.1 uses the constant 0 instead.

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

Designates C function *f* as the function Lua calls when an unexpected error occurs, and returns the previously designated panic function. When *f* is called, the error message is at the top of the stack.

After the panic function returns, Lua aborts the host application. This unhappy outcome can be avoided if the host performs a longjmp() of its own to recover.

Retrieve Error Information

In addition to having the error message at the top of the stack after an error occurred, the host can also retrieve a traceback with additional error information.

```
void luaL_traceback(lua_State *L, lua_State *L1,
                    const char *message,
                    int level);
```

Lua 5.2, 5.3

Pushes onto the stack a string traceback of the call stack in thread *L1* at call level number *level*, with optional string message *message* prepended to the traceback. A *level* of `0` is the current function (or the current file or module if there is no current function), `1` is the function that called the current function, `2` is the caller of the function that called the current function, and so on.

Load and Run Dynamic Code

The host can load and execute user-provided chunks of Lua code at run-time. It can also do this in a sandboxed environment as a security measure. Example 31 illustrates how the host can run user-defined Lua scripts in a tightly-controlled environment that does not provide access to external modules, the underlying filesystem and operating system, and any other potentially unsafe Lua features.

Example 31. Run user-defined Lua code in a sandbox

```
// Define and store the sandbox for subsequent use.
const char *safe[] = {
  "assert", "error", "ipairs", "math", "next", "pairs",
  "pcall", "select", "string", "table", "tonumber",
  "tostring", "type", "xpcall", NULL
};
lua_newtable(L); // the sandbox environment
for (const char **p = safe; *p; p++)
  lua_getglobal(L, *p), lua_setfield(L, -2, *p);
/* add other safe host functions to sandbox... */
int sandbox_ref = luaL_ref(L, LUA_REGISTRYINDEX);

/* ... */

// Attempt to load the user-defined Lua script
// (text-only) as an anonymous function.
if (luaL_loadfilex(L, user_script, "t") == LUA_OK) {
  // Make the sandbox the function's environment.
  lua_rawgeti(L, LUA_REGISTRYINDEX, sandbox_ref);
  lua_setupvalue(L, -2, 1);
  // Execute the script.
  if (lua_pcall(L, 0, 0, 0) != LUA_OK) {
    /* process and pop error message at index -1... */
  }
}
```

```
/* ... */

// Finished with the sandbox; delete it.
luaL_unref(L, LUA_REGISTRYINDEX, sandbox_ref);
```

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

Executes the contents of string *s* or the file identified by string *filename* as a chunk of Lua code, pushes onto the stack all values returned by that chunk, and returns `LUA_OK` (or `0` in Lua 5.1) on success.

If an error occurred, a non-zero error code is returned and the error message is pushed onto the stack instead. In addition to the error codes listed in Table 15 on page 123, `LUA_ERRSYNTAX` and `LUA_ERRFILE` can also be returned, which indicate there was a syntax error or problem opening the file, respectively.

```
int luaL_loadstring (lua_State *L, const char *s);
int luaL_loadbuffer (lua_State *L, const char *s,
                     size_t len, const char *name);
int luaL_loadbufferx(lua_State *L, const char *s,
                     size_t len, const char *name,
                     const char *mode);           Lua 5.2, 5.3
int luaL_loadfile   (lua_State *L, const char *filename);
int luaL_loadfilex  (lua_State *L, const char *filename,
                     const char *mode);           Lua 5.2, 5.3
```

Loads as a chunk of Lua code zero-terminated string *s*, string *s* of length *len* bytes, or the contents of the file identified by string *filename*, pushes onto the stack a Lua function that will execute that chunk when called, and returns `LUA_OK` (or `0` in Lua 5.1) on success. *name* is an optional string name associated with the chunk and *mode* indicates whether the chunk can be text (`"t"`), binary (`"b"`), or both (`"bt"`). The default value of *mode* is `"bt"`. (Binary chunks are produced by Lua's *luac* or *luac.exe* executable.)

If an error occurred, a non-zero error code is returned and the error message is pushed onto the stack instead. In addition to the error codes listed in Table 15 on page 123, `LUA_ERRSYNTAX` and `LUA_ERRFILE` can also be returned. The former indicates there was a syntax error and the latter indicates there was a problem opening the file.

CAUTION

Lua does not verify the integrity of, or in any way sanitize binary chunks. Running truly arbitrary binary chunks may be unsafe.

```
const char *lua_setupvalue(lua_State *L,
                           int index, 1);          Lua 5.2, 5.3
int lua_setfenv(lua_State *L, int index);          Lua 5.1
```

Pops a table value off the stack, designates it as the environment of the function value at stack index *`index`*, and returns non-`NULL` or `1` on success. The section "Environments" on page 44 describes environments.

Threading in C

The host can create and use threads similarly to how Lua can create and use threads as illustrated in the section "Thread Facilities" on page 65. However, the typical threading procedure in C differs slightly from the threading procedure in Lua:

1. The main Lua thread creates and pushes a new (suspended) thread *T* onto the stack.
2. The main thread pushes a function body onto the stack of *T*. (*T* has its own stack, but shares the same global environment.)
3. Upon starting *T*, the main thread is temporarily suspended, and the body of *T* is executed.
4. *T* performs some work and then yields back to the main thread.
5. The main thread resumes right where it left off, at the point where it started *T*. *T* is now suspended.
6. The main thread performs some work and then resumes *T*.
7. *T* resumes, but not right where it left off (at the point where it yielded back to the main thread). Instead, *T* either resumes in the caller of the function that yielded *T*, or resumes in the *continuation function* specified by the function that yielded *T*. The main thread is now suspended.
8. This process repeats until *T* completes its work and the

thread finishes.

9. The main thread resumes right where it left off and continues indefinitely. *T* is now dead and cannot be resumed.

During each transition between threads, values can be exchanged between the thread stacks. When the main thread starts *T*, it can pass values from its stack to the function body of *T*. When *T* yields, it can pass values from its stack back to the main thread's stack. When the main thread resumes *T*, it can pass more of its stack values to *T*. And so on.

Example 32 illustrates the entirety of this typical threading procedure by starting a series of threads that continuously monitor files for output, by having those threads pass that output back to the main thread for processing, and by having the main thread ask monitoring threads to stop monitoring based on their processed output. (While this example operates on files, it can be adapted to work on other resources like sockets and pipes.)

In addition to the typical threading procedure, there is a quirk involving a C function that calls another function that eventually yields. This case is also handled using continuation functions.

All of the aforementioned aspects of threading in C, including continuation functions, are described in the following sections.

Example 32. Monitor output from a set of files

```
// Filenames to monitor.
const char *filenames[32]; // should have NULL sentinel

/* ... */

// Thread body continuation function for monitoring a
// file.
static int l_monitork(lua_State *thread, int status,
                      lua_KContext ctx) {
  FILE *f = (FILE*)ctx;
  // Stop monitoring file if requested to.
  if (status == LUA_YIELD &&
      !lua_toboolean(thread, 1)) {
    fclose(f);
```

```
    return 0;
  }
  // Check for data to be read.
  int c = getc(f);
  if (c != EOF) {
    // Read and yield a line of data.
    ungetc(c, f);
    char buf[BUFSIZ];
    fgets(buf, BUFSIZ, f);
    lua_pushstring(thread, buf);
    return lua_yieldk(thread, 1, ctx, l_monitork);
  } else {
    // No data to read; yield nothing.
    return lua_yieldk(thread, 0, ctx, l_monitork);
  }
}

// Thread body function for monitoring a file.
static int l_monitorfile(lua_State *thread) {
  const char *filename = luaL_checkstring(thread, 1);
  FILE *f = fopen(filename, "r");
  if (!f)
    return luaL_error(thread, "file '%s' not found",
                      filename);
  return l_monitork(thread, LUA_OK, (lua_KContext)f);
}

/* ... */

// Create and start threads.
lua_createtable(L, 32, 0); // active threads table
for (int i = 0; i < 32; i++) {
  if (!filenames[i]) break;
  lua_State *thread = lua_newthread(L);
  lua_pushcfunction(thread, l_monitorfile);
  lua_pushstring(thread, filenames[i]);
  if (lua_resume(thread, L, 1) == LUA_YIELD)
    // Store thread for monitoring.
    lua_rawseti(L, -2, lua_rawlen(L, -2) + 1);
  else {
    /* handle error starting thread... */
    lua_pop(L, 1);
  }
}
```

```
// Monitor active threads.
int i = 1;
while (lua_rawlen(L, -1) > 0) {
  lua_rawgeti(L, -1, i);
  lua_State *thread = lua_tothread(L, -1);
  if (lua_gettop(thread) > 0) {
    // Thread has output from its monitored file.
    const char *line = lua_tostring(thread, 1);
    /* process line and possibly stop monitoring... */
    lua_pushboolean(thread, keep_monitoring);
    lua_replace(thread, 1);
    lua_resume(thread, L, 1);
    if (!keep_monitoring) {
      // Stop monitoring the now-dead thread.
      lua_getglobal(L, "table");
      lua_getfield(L, -1, "remove");
      lua_replace(L, -2);
      lua_pushvalue(L, -3); // active threads table
      lua_pushnumber(L, i);
      lua_call(L, 2, 0); // table.remove(threads, i)
      lua_pop(L, 1); // dead thread
      continue; // monitor next thread
    }
  }
  lua_pop(L, 1); // thread
  if (++i > lua_rawlen(L, -1)) i = 1; // start again
}
lua_pop(L, 1); // active threads table
```

Create a Thread

The host can create threads. Each thread has its own stack for pushing values onto (such as a function body) and popping values off of (such as return values).

`lua_State`
: A C `struct` that represents both a thread in a Lua interpreter and the interpreter itself.

`lua_State *lua_newthread(lua_State *L);`
: Creates and pushes onto the stack a new (suspended) thread and returns a pointer to it. The new thread has its own stack, but shares the same global environment as Lua interpreter *L*.

Start or Resume a Thread

The host can start a thread that has a function body on its stack and can resume a thread that had previously yielded.

`void lua_pushcfunction(lua_State *thread, lua_CFunction f);`
: Pushes C function value *f* onto the stack of thread *thread*.

`int lua_resume(lua_State *thread, lua_State *L, int nargs);` **Lua 5.2, 5.3**
`int lua_resume(lua_State *thread, int nargs);` **Lua 5.1**
: Starts or resumes execution of thread *thread* from thread *L* (the currently active thread), and returns `LUA_OK` (or `0` in Lua 5.1) if *thread* finishes without error, `LUA_YIELD` if *thread* subsequently yields, or a non-zero error code if *thread* raises an error. Table 15 on page 123 lists Lua's error codes and their meanings.

When starting *thread*, *nargs* function argument values are popped off the stack of *thread*, the thread function body now at the top of the stack is popped off the stack, and that popped function is called with the popped arguments (the last value popped being the first argument and the first value popped being the last argument).

When resuming *thread*, all values on its stack are either left for the continuation function passed to the call to `lua_yieldk()` that originally yielded *thread*, left for the caller of `lua_yield()`, or used as the return values of the yielding call to `coroutine.yield()`.

If *thread* subsequently yields without error, the only values on its stack are the argument values specified by the yielding call. If *thread* finishes without error, the only values on its stack are the values returned by the function body of *thread*. If *thread* raises an error, the error message is at the top of its stack.

Yield a Thread

The host can yield the running thread. After a C function yields, it is impossible to return to that function when the thread resumes, due to the nature of the yield. Instead, Lua

5.2 and 5.3 allow for a continuation function to be called upon resumption. Lua 5.1 simply returns to the caller of the C function.

`int lua_isyieldable(lua_State *L, int index);` **Lua 5.3**

Returns `1` if the value at stack index ***index*** is a yieldable thread. Otherwise, returns `0`.

`lua_KFunction` **Lua 5.3**

The C type associated with continuation functions:

```
typedef int (*lua_KFunction) (lua_State *thread,
                              int status,
                              lua_KContext ctx);
```

When a continuation function is called by Lua, `status` is `LUA_YIELD`. When calling a continuation function manually, `status` is either `LUA_OK`, or the non-zero error code returned by `lua_pcallk()` if an error occurred.

`lua_KContext` **Lua 5.3**

The C type associated with continuation function contexts (typically `intptr_t` or `ptrdiff_t`, which are large enough to store an arbitrary pointer). Continuation function contexts are unused by Lua, but may be useful to the host for passing around state information.

`int lua_getctx(lua_State *thread, int *ctx);` **Lua 5.2**

Returns `LUA_YIELD` if the current function was called by Lua to continue from a yield, and sets ***ctx*** to the value passed to the call to `lua_yieldk()` that yielded thread ***thread***. Otherwise, returns `LUA_OK` and leaves ***ctx*** unmodified.

`int lua_yieldk(lua_State *thread, int nresults, lua_KContext ctx, lua_KFunction k);` **Lua 5.3**

`int lua_yieldk(lua_State *thread, int nresults, int ctx, lua_CFunction k);` **Lua 5.2**

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

Yields thread ***thread***, and either leaves only the top ***nresults*** values on its stack for use by the `lua_resume()` call that originally started or resumed ***thread***, or uses those values as the (potentially extra) return values of the Lua call that originally started or resumed ***thread***.

When ***thread*** is resumed again, ***k(thread***, `LUA_YIELD`, *ctx*) is called (or just *k*(*thread*) in Lua 5.2), and the only values on the stack of *thread* are either the values left by the resuming `lua_resume()` call, or the argument values passed to the resuming Lua call. If there is no continuation function (which is always the case in Lua 5.1), execution returns to the original caller of this function.

Transfer Values Between Threads

When a thread yields, the host can transfer that thread's stack values to the main thread (or any other live thread). Similarly, the host can transfer values from the stack of another live thread to the thread about to be resumed.

`void lua_xmove(lua_State *from, lua_State *to, int n);`
: Pops *n* values off the stack of thread *from* and pushes them onto the stack of thread *to*. Both *from* and *to* must share the same Lua interpreter.

Query a Thread's Status

The host can query the status of a thread.

`int lua_status(lua_State *thread);`
: Returns the status of thread *thread*: `LUA_OK` (or `0` in Lua 5.1) for a normal thread (active but not running, not yet started, or finished without error), `LUA_YIELD` if *thread* has yielded, or the non-zero error code returned by `lua_resume()` if *thread* raised an error. Table 15 on page 123 lists Lua's error codes and their meanings.

Call a Function that Yields

Any running thread (including the main thread) can invoke functions, including C functions. These C functions can in turn invoke other functions. A potential problem may arise if a C function *f* invokes another function that ultimately yields. When the suspended thread resumes, it is impossible to return to *f* due to the nature of the yield. While Lua 5.1 will throw an error at this attempt to "yield across a C-call boundary," Lua 5.2 and 5.3 allow for a continuation function to be called upon resumption. Example 33 demonstrates how to handle this case as it iterates over all key-value pairs in a table and calls a potentially yielding function with each pair as arguments.

Example 33. Call a function for each table key-value pair

```
// Thread body continuation function for iterating over
// a table's key-value pairs and calling a function
// with each pair as that function's arguments.
static int l_iteratek(lua_State *thread, int status,
                      lua_KContext ctx) {
  if (status == LUA_OK)
    lua_pushnil(thread); // start iteration
  else
    lua_pop(thread, 1); // previous value
  while (lua_next(thread, 1) != 0) {
    lua_pushvalue(thread, lua_upvalueindex(1));
    lua_pushvalue(thread, -3); // key
    lua_pushvalue(thread, -3); // value
    lua_callk(thread, 2, 0, 0, l_iteratek);
    lua_pop(thread, 1); // value
  }
  return 0;
}

// Initial thread body function.
static int l_iterate(lua_State *thread) {
  return l_iteratek(thread, LUA_OK, 0);
}

/* ... */

lua_State *thread = lua_newthread(L);
/* push function to be called each iteration... */
lua_pushcclosure(thread, l_iterate, 1);
/* push table to be iterated over... */
while (lua_resume(thread, L, 1) == LUA_YIELD) {
  /* work to do in-between yields... */
}
lua_pop(L, 1); // dead thread
```

`lua_KFunction` **Lua 5.3**

The C type associated with continuation functions:

```
typedef int (*lua_KFunction) (lua_State *thread,
                              int status,
                              lua_KContext ctx);
```

When a continuation function is called by Lua, `status` is `LUA_YIELD`. When calling a continuation function manually, `status` is either `LUA_OK`, or the non-zero error code returned by `lua_pcallk()` if an error occurred.

`lua_KContext` **Lua 5.3**

The C type associated with continuation function contexts (typically `intptr_t` or `ptrdiff_t`, which are large enough to store an arbitrary pointer). Continuation function contexts are unused by Lua, but may be useful to the host for passing around state information.

`int lua_getctx(lua_State *thread, int *ctx);` **Lua 5.2**

Returns `LUA_YIELD` if the current function was called by Lua to continue from a yield, and sets ***`ctx`*** to the value passed to the call to `lua_yieldk()` that yielded thread ***`thread`***. If the current function was called by Lua after an error occurred in a call to `lua_pcallk()`, returns a non-zero error code and sets ***`ctx`*** to the value passed to `lua_pcallk()`. Otherwise, returns `LUA_OK` and leaves ***`ctx`*** unmodified. Table 15 on page 123 lists Lua's error codes and their meanings.

```
void lua_callk(lua_State *thread, int nargs, int nresults,
               lua_KContext ctx, lua_KFunction k);      Lua 5.3
void lua_callk(lua_State *thread, int nargs, int nresults,
               int ctx, lua_CFunction k);               Lua 5.2
int lua_pcallk(lua_State *thread, int nargs, int nresults,
               int error_handler, lua_KContext ctx,
               lua_KFunction k);                        Lua 5.3
int lua_pcallk(lua_State *thread, int nargs, int nresults,
               int error_handler, int ctx,
               lua_CFunction k);                        Lua 5.2
```

Pops ***`nargs`*** function argument values off the stack, pops off the stack the function that is now at the top of the stack, calls that popped function with the popped arguments (the last value popped being the first argument and the first value popped being the last argument), and pushes the first ***`nresults`*** values returned by the function onto the stack (or all of them if ***`nresults`*** is `LUA_MULTRET`). `lua_pcallk()` returns `LUA_OK` on success.

`lua_callk()` should only be called from within C functions that do not care to handle errors and have been ultimately invoked by a protected call. The section "Error

Handling" on page 122 describes protected calls.

If an error occurs, `lua_pcallk()` pushes the error message onto the stack and returns a non-zero error code. If *error_handler* is nonzero, the function at stack index *error_handler* is called with the error message as an argument, and that function's return value is the error message ultimately pushed onto the stack. Table 15 on page 123 lists Lua's error codes and their meanings.

If *thread* yields during the call, the original `lua_callk()` or `lua_pcallk()` call will not return. Instead, whenever *thread* resumes, `k(thread, LUA_YIELD, ctx)` is called (or just `k(thread)` in Lua 5.2), and the stack contains the first *nresults* values returned by the originally called function (or all of the returned values if *nresults* is `LUA_MULTRET`).

If the value being called is a table or userdata value with the metamethod `__call()`, that metamethod is called to perform the operation. The section "Other Operator and Statement Metamethods" on page 34 covers this metamethod in its generic form.

Memory Management

Lua manages the memory of its values by allocating memory for new values and freeing memory for values no longer in use. Lua employs a garbage collector to automatically detect and delete unused values. More often than not this is sufficient. However, Lua provides access controls for its collector should the need arise.

`int lua_gc(lua_State *L, LUA_GCCOLLECT, 0);`
: Performs a full garbage collection cycle.

`int lua_gc(lua_State *L, LUA_GCSTOP, 0);`
`int lua_gc(lua_State *L, LUA_GCRESTART, 0);`
: Stops and restarts automatic garbage collection.

`int lua_gc(lua_State *L, LUA_GCISRUNNING, 0);`
: Returns `1` if automatic garbage collection is on and `0` if it is off.

`int lua_gc(lua_State *L, LUA_GCCOUNT, 0);`
: Returns the number of kilobytes of memory used by Lua.

Miscellaneous

Lua provides other miscellaneous C API facilities.

`int lua_numbertointeger(lua_Number n, lua_Integer *i);` **Lua 5.3**

Converts float *n* to an integer, stores the result in *i*, and returns `1` or, if the conversion fails, returns `0`.

```
size_t lua_stringtonumber(lua_State *L,
                          const char *s);
```
Lua 5.3

Converts string *s* to a number and, if successful, pushes that number onto the stack and returns a number greater than zero. A return value of `0` indicates the conversion failed and that nothing was pushed.

`void lua_pushglobaltable(lua_State *L);` **Lua 5.2, 5.3**
`lua_pushvalue(L, LUA_GLOBALSINDEX);` **Lua 5.1**

Pushes the global environment table onto the stack.

`int  lua_getuservalue(lua_State *L, int index);` **Lua 5.3**
`void lua_getuservalue(lua_State *L, int index);` **Lua 5.2**

Pushes onto the stack the Lua value associated with the full userdata value at stack index *index*.

Retrieving Lua values directly associated with userdata is more efficient than a registry table lookup.

`lua_setuservalue(lua_State *L, int index);` **Lua 5.2, 5.3**

Pops a value off the stack and associates it with the full userdata value at stack index *index*.

In Lua 5.2, the popped value must be a table.

`void lua_getfenv(lua_State *L, int index);` **Lua 5.1**

Pushes onto the stack the environment table of the function, thread, or userdata value at stack index *index*.

Lua and C API Index

Symbols

Keywords

_G

luaL_*