Programming in Lua, Fourth Edition

Roberto Ierusalimschy

Part IV. The C API

Table of Contents

27. An Overview of the C API 223
- A First Example 223
- The Stack 225
  - Pushing elements 226
  - Querying elements 227
  - Other stack operations 229
- Error Handling with the C API 231
  - Error handling in application code 232
  - Error handling in library code 232
- Memory Allocation 233

28. Extending Your Application 236
- The Basics 236
- Table Manipulation 237
  - Some short cuts 240
- Calling Lua Functions 241
- A Generic Call Function 242

29. Calling C from Lua 247
- C Functions 247
- Continuations 249
- C Modules 251

30. Techniques for Writing C Functions 254
- Array Manipulation 254
- String Manipulation 255
- Storing State in C Functions 258
  - The registry 258
  - Upvalues 260
  - Shared upvalues 263

31. User-Defined Types in C 265
- Userdata 265
- Metatables 268
- Object-Oriented Access 270
- Array Access 271
- Light Userdata 272

32. Managing Resources 274
- A Directory Iterator 274
- An XML Parser 277

33. Threads and States 286
- Multiple Threads 286
- Lua States 289

Chapter 27. An Overview of the C API

Lua is an *embedded language*. This means that Lua is not a stand-alone application, but a library that we can link with other applications to incorporate Lua facilities into them.

You may be wondering: if Lua is not a stand-alone program, how come we have been using Lua stand-alone through the whole book until now? The solution to this puzzle is the Lua interpreter —the executable `lua`. This executable is a small application, around six hundred lines of code, that uses the Lua library to implement the stand-alone interpreter. The program handles the interface with the user, taking her files and strings to feed them to the Lua library, which does the bulk of the work (such as actually running Lua code).

This ability to be used as a library to extend an application is what makes Lua an *embeddable language*. At the same time, a program that uses Lua can register new functions in the Lua environment; such functions are implemented in C (or another language), so that they can add facilities that cannot be written directly in Lua. This is what makes Lua an *extensible language*.

These two views of Lua (as an embeddable language and as an extensible language) correspond to two kinds of interaction between C and Lua. In the first kind, C has the control and Lua is the library. The C code in this kind of interaction is what we call *application code*. In the second kind, Lua has the control and C is the library. Here, the C code is called *library code*. Both application code and library code use the same API to communicate with Lua, the so-called C API.

The C API is the set of functions, constants, and types that allow C code to interact with Lua.[1] It comprises functions to read and write Lua global variables, to call Lua functions, to run pieces of Lua code, to register C functions so that they can later be called by Lua code, and so on. Virtually anything that Lua code can do can also be done by C code through the C API.

The C API follows the *modus operandi* of C, which is quite different from that of Lua. When programming in C, we must care about type checking, error recovery, memory-allocation errors, and several other sources of complexity. Most functions in the API do not check the correctness of their arguments; it is our responsibility to make sure that the arguments are valid before calling a function.[2] If we make mistakes, we can get a crash instead of a well-behaved error message. Moreover, the API emphasizes flexibility and simplicity, sometimes at the cost of ease of use. Common tasks may involve several API calls. This may be boring, but it gives us full control over all details.

As its title says, the goal of this chapter is to give an overview of what is involved when we use Lua from C. Do not try to understand all the details of what is going on now; we will fill them in later. Nevertheless, do not forget that you always can find more details about specific functions in the Lua reference manual. Moreover, you can find several examples of API uses in the Lua distribution itself. The Lua stand-alone interpreter (`lua.c`) provides examples of application code, while the standard libraries (`lmathlib.c`, `lstrlib.c`, etc.) provide examples of library code.

From now on, we are wearing a C programmer's hat.

A First Example

We will start this overview with a simple example of an application program: a stand-alone Lua interpreter. We can write a bare-bones stand-alone interpreter as in Figure 27.1, “A bare-bones stand-alone Lua interpreter”.

[1] Throughout this text, the term “function” actually means “function or macro”. The API implements several facilities as macros.

[2] You can compile Lua with the macro `LUA_USE_APICHECK` defined to enable some checks; this option is particularly useful when debugging your C code. Nevertheless, several errors simply cannot be detected in C, such as invalid pointers.

Figure 27.1. A bare-bones stand-alone Lua interpreter

```
#include <stdio.h>
#include <string.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (void) {
  char buff[256];
  int error;
  lua_State *L = luaL_newstate();             /* opens Lua */
  luaL_openlibs(L);        /* opens the standard libraries */

  while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadstring(L, buff) || lua_pcall(L, 0, 0, 0);
    if (error) {
      fprintf(stderr, "%s\n", lua_tostring(L, -1));
      lua_pop(L, 1);  /* pop error message from the stack */
    }
  }

  lua_close(L);
  return 0;
}
```

The header file `lua.h` declares the basic functions provided by Lua. It includes functions to create a new Lua environment, to invoke Lua functions, to read and write global variables in the environment, to register new functions to be called by Lua, and so on. Everything declared in `lua.h` has a `lua_` prefix (e.g., `lua_pcall`).

The header file `lauxlib.h` declares the functions provided by the *auxiliary library* (auxlib). All its declarations start with `luaL_` (e.g., `luaL_loadstring`). The auxiliary library uses the basic API provided by `lua.h` to provide a higher abstraction level, in particular with abstractions used by the standard libraries. The basic API strives for economy and orthogonality, whereas the auxiliary library strives for practicality for a few common tasks. Of course, it is very easy for your program to create other abstractions that it needs, too. Keep in mind that the auxiliary library has no access to the internals of Lua. It does its entire job through the official basic API declared in `lua.h`. Whatever it does, your program can do too.

The Lua library defines no C global variables at all. It keeps all its state in the dynamic structure `lua_State`; all functions inside Lua receive a pointer to this structure as an argument. This design makes Lua reentrant and ready to be used in multithreaded code.

As its name implies, the function `luaL_newstate` creates a new Lua state. When `luaL_newstate` creates a fresh state, its environment contains no predefined functions, not even `print`. To keep Lua small, all standard libraries come as separate packages, so that we do not have to use them if we do not need to. The header file `lualib.h` declares functions to open the libraries. The function `luaL_openlibs` opens all standard libraries.

After creating a state and populating it with the standard libraries, it is time to handle user input. For each line the user enters, the program first compiles it with `luaL_loadstring`. If there are no errors, the call returns zero and pushes the resulting function on the stack. (We will discuss this mysterious stack in the next section.) Then the program calls `lua_pcall`, which pops the function from the stack and runs it in protected mode. Like `luaL_loadstring`, `lua_pcall` returns zero if there are no errors. In case of error, both functions push an error message on the stack; we then get this message with `lua_tostring` and, after printing it, remove it from the stack with `lua_pop`.

Real error handling can be quite complex in C, and how to do it depends on the nature of our application. The Lua core never writes anything directly to any output stream; it signals errors by returning error messages. Each application can handle these messages in a way most appropriate to its needs. To simplify our discussions, we will assume for our next examples a simple error handler like the following one, which prints an error message, closes the Lua state, and finishes the whole application:

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void error (lua_State *L, const char *fmt, ...) {
  va_list argp;
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  lua_close(L);
  exit(EXIT_FAILURE);
}
```

Later we will discuss more about error handling in the application code.

Because we can compile Lua as either C or C++ code, `lua.h` does not include the following boilerplate commonly used in C libraries:

```
#ifdef __cplusplus
extern "C" {
#endif
   ...
#ifdef __cplusplus
}
#endif
```

If we have compiled Lua as C code and are using it in C++, we can include `lua.hpp` instead of `lua.h`. It is defined as follows:

```
extern "C" {
#include "lua.h"
}
```

The Stack

A major component in the communication between Lua and C is an omnipresent virtual *stack*. Almost all API calls operate on values on this stack. All data exchange from Lua to C and from C to Lua occurs through this stack. Moreover, we can use the stack to keep intermediate results, too.

We face two problems when trying to exchange values between Lua and C: the mismatch between a dynamic and a static type system and the mismatch between automatic and manual memory management.

In Lua, when we write `t[k] = v`, both `k` and `v` can have several different types; even `t` can have different types, due to metatables. If we want to offer this operation in C, however, any given `settable` function must have a fixed type. We would need dozens of different functions for this single operation (one function for each combination of types for the three arguments).

We could solve this problem by declaring some kind of union type in C —let us call it `lua_Value`— that could represent all Lua values. Then, we could declare `settable` as

```
    void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

This solution has two drawbacks. First, it can be difficult to map such a complex type to other languages; we designed Lua to interface easily not only with C/C++, but also with Java, Fortran, C#, and the like. Second, Lua does garbage collection: if we keep a Lua table in a C variable, the Lua engine has no way to know about this use; it may (wrongly) assume that this table is garbage and collect it.

Therefore, the Lua API does not define anything like a `lua_Value` type. Instead, it uses the stack to exchange values between Lua and C. Each slot in this stack can hold any Lua value. Whenever we want to ask for a value from Lua (such as the value of a global variable), we call Lua, which pushes the required value onto the stack. Whenever we want to pass a value to Lua, we first push the value onto the stack, and then we call Lua (which will pop the value). We still need a different function to push each C type onto the stack and a different function to get each C type from the stack, but we avoid combinatorial explosion. Moreover, because this stack is part of the Lua state, the garbage collector knows which values C is using.

Nearly all functions in the API use the stack. As we saw in our first example, `luaL_loadstring` leaves its result on the stack (either the compiled chunk or an error message); `lua_pcall` gets the function to be called from the stack and leaves any error message there too.

Lua manipulates this stack in a strict LIFO discipline (Last In, First Out). When we call Lua, it changes only the top part of the stack. Our C code has more freedom; specifically, it can inspect any element in the stack and even insert and delete elements at any position.

Pushing elements

The API has a push function for each Lua type with a direct representation in C: `lua_pushnil` for the constant nil, `lua_pushboolean` for Booleans (integers, in C), `lua_pushnumber` for doubles,[3] `lua_pushinteger` for integers, `lua_pushlstring` for arbitrary strings (a pointer to `char` plus a length), and `lua_pushstring` for zero-terminated strings:

```
    void lua_pushnil      (lua_State *L);
    void lua_pushboolean  (lua_State *L, int bool);
    void lua_pushnumber   (lua_State *L, lua_Number n);
    void lua_pushinteger  (lua_State *L, lua_Integer n);
    void lua_pushlstring  (lua_State *L, const char *s, size_t len);
    void lua_pushstring   (lua_State *L, const char *s);
```

There are also functions to push C functions and userdata values onto the stack; we will discuss them later.

SMALL LUA

The type `lua_Number` is the numeric float type in Lua. It is `double` by default, but we can configure Lua at compile time to use `float` or even `long double`. The type `lua_Integer` is the numeric integer type in Lua. Usually, it is defined as `long long`, which is a signed 64-bit integer. Again, it is trivial to configure Lua to use `int` or `long` for this type. The combination `float–int`, with 32-bit floats and integers, creates what we call Small Lua, which is particularly interesting for small machines and restricted hardware.[4]

[3] For historical reasons, the term “number” in the API refers to doubles.

[4] For these configurations, have a look in the file `luaconf.h`.

!!! LUA STRINGS ARE NOT ZERO-TERMINATED

Strings in Lua are not zero-terminated; they can contain arbitrary binary data. In consequence, the basic function to push a string onto the stack is `lua_pushlstring`, which requires an explicit length as an argument. For zero-terminated strings, we can use also `lua_pushstring`, which uses `strlen` to supply the string length. Lua never keeps pointers to external strings (or to any other external object except C functions, which are always static). For any string that it has to keep, Lua either makes an internal copy or reuses one. Therefore, we can free or modify our buffers as soon as these functions return.

Whenever we push an element onto the stack, it is our responsibility to ensure that the stack has space for it. Remember, you are a C programmer now; Lua will not spoil you. When Lua starts and any time that Lua calls C, the stack has at least 20 free slots. (The header file `lua.h` defines this constant as `LUA_MINSTACK`.) This space is more than enough for most common uses, so usually we do not even think about it. However, some tasks need more stack space, in particular if we have a loop pushing elements onto the stack. In those cases, we need to call `lua_checkstack`, which checks whether the stack has enough space for our needs:

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

Here, `sz` is the number of extra slots we need. If possible, `lua_checkstack` grows the stack to accommodate the required extra size. Otherwise, it returns zero.

The auxiliary library offers a higher-level function to check for stack space:

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

This function is similar to `lua_checkstack` but, if it cannot fulfill the request, it raises an error with the given message, instead of returning an error code.

Querying elements

NATURAL TO MEASURE FROM TOP (1) OR BOTTOM (-1)

To refer to elements on the stack, the API uses *indices*. The first element pushed on the stack has index 1, the next one has index 2, and so on. We can also access elements using the top of the stack as our reference, with negative indices. In this case, -1 refers to the element on top (that is, the last element pushed), -2 to the previous element, and so on. For instance, the call `lua_tostring(L, -1)` returns the value on the top of the stack as a string. As we will see, there are several occasions when it is natural to index the stack from the bottom (that is, with positive indices), and several other occasions when the natural way is to use negative indices.

To check whether a stack element has a specific type, the API offers a family of functions called `lua_is*`, where the `*` can be any Lua type. So, there are `lua_isnil`, `lua_isnumber`, `lua_isstring`, `lua_istable`, and the like. All these functions have the same prototype:

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

Actually, `lua_isnumber` does not check whether the value has that specific type, but whether the value can be converted to that type; `lua_isstring` is similar: in particular, any number satisfies `lua_isstring`.

There is also a function `lua_type`, which returns the type of an element on the stack. Each type is represented by a respective constant: `LUA_TNIL`, `LUA_TBOOLEAN`, `LUA_TNUMBER`, `LUA_TSTRING`, etc. We use this function mainly in conjunction with a switch statement. It is also useful when we need to check for strings and numbers without potential coercions.

To get a value from the stack, there are the `lua_to*` functions:

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

```
const char  *lua_tolstring (lua_State *L, int index,
                                           size_t *len);
lua_State   *lua_tothread (lua_State *L, int index);
lua_Number   lua_tonumber  (lua_State *L, int index);
lua_Integer  lua_tointeger (lua_State *L, int index);
```

We can call any of these functions even when the given element does not have an appropriate type. The function `lua_toboolean` works for any type, converting any Lua value to a C Boolean according to the Lua rules for conditions: zero for the values nil and **false**, and one for any other Lua value. The functions `lua_tolstring` and `lua_tothread` return `NULL` for values with incorrect types. The numeric functions, however, have no way to signal a wrong type, so they simply return zero. Formerly we would need to call `lua_isnumber` to check the type, but Lua 5.2 introduced the following new functions:

```
lua_Number   lua_tonumberx (lua_State *L, int idx, int *isnum);
lua_Integer  lua_tointegerx (lua_State *L, int idx, int *isnum);
```

The out parameter `isnum` returns a Boolean that indicates whether the Lua value was successfully coerced to the desired type.

The function `lua_tolstring` returns a pointer to an internal copy of the string and stores the string's length in the position given by `len`. We must not change this internal copy (there is a `const` there to remind us). Lua ensures that this pointer is valid as long as the corresponding string value is on the stack. When a C function called by Lua returns, Lua clears its stack; therefore, as a rule, we should never store pointers to Lua strings outside the function that got them.

Any string that `lua_tolstring` returns always has an extra zero at its end, but it can have other zeros inside it. The size returned through the third argument, `len`, is the real string's length. In particular, assuming that the value on the top of the stack is a string, the following assertions are always valid:

```
size_t len;
const char *s = lua_tolstring(L, -1, &len); /* any Lua string */
assert(s[len] == '\0');
assert(strlen(s) <= len);
```

We can call `lua_tolstring` with `NULL` as its third argument if we do not need the length. Better yet, we can use the macro `lua_tostring`, which simply calls `lua_tolstring` with a `NULL` third argument.

To illustrate the use of these functions, Figure 27.2, “Dumping the stack” presents a useful helper function that dumps the entire content of the stack.

Figure 27.2. Dumping the stack

```
static void stackDump (lua_State *L) {
  int i;
  int top = lua_gettop(L);  /* depth of the stack */
  for (i = 1; i <= top; i++) {  /* repeat for each level */
    int t = lua_type(L, i);
    switch (t) {
      case LUA_TSTRING: {  /* strings */
        printf("'%s'", lua_tostring(L, i));
        break;
      }
      case LUA_TBOOLEAN: {  /* Booleans */
        printf(lua_toboolean(L, i) ? "true" : "false");
        break;
      }
      case LUA_TNUMBER: {  /* numbers */
        printf("%g", lua_tonumber(L, i));
        break;
      }
      default: {  /* other values */
        printf("%s", lua_typename(L, t));
        break;
      }
    }
    printf("  ");  /* put a separator */
  }
  printf("\n");  /* end the listing */
}
```

This function traverses the stack from bottom to top, printing each element according to its type. It prints strings between quotes; for numbers it uses a `"%g"` format; for values with no C equivalents (tables, functions, etc.), it prints only their types. (`lua_typename` converts a type code to a type name.)

In Lua 5.3, we can still print all numbers with `lua_tonumber` and the `"%g"` format, as integers are always coercible to floats. However, we may prefer to print integers as integers, to avoid losing precision. In that case, we can use the new function `lua_isinteger` to distinguish integers from floats:

```
      case LUA_TNUMBER: {  /* numbers */
        if (lua_isinteger(L, i))  /* integer? */
          printf("%lld", lua_tointeger(L, i));
        else  /* float */
          printf("%g", lua_tonumber(L, i));
        break;
      }
```

Other stack operations

Besides the previous functions, which exchange values between C and the stack, the API offers also the following operations for generic stack manipulation:

```
int  lua_gettop    (lua_State *L);
void lua_settop    (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_rotate    (lua_State *L, int index, int n);
void lua_remove    (lua_State *L, int index);
void lua_insert    (lua_State *L, int index);
void lua_replace   (lua_State *L, int index);
void lua_copy      (lua_State *L, int fromidx, int toidx);
```

The function `lua_gettop` returns the number of elements on the stack, which is also the index of the top element. The function `lua_settop` sets the top (that is, the number of elements on the stack) to a specific value. If the previous top was higher than the new one, the function discards the extra top values. Otherwise, it pushes nils on the stack to get the given size. In particular, `lua_settop(L, 0)` empties the stack. We can also use negative indices with `lua_settop`. Using this facility, the API offers the following macro, which pops `n` elements from the stack:

```
#define lua_pop(L,n)  lua_settop(L, -(n) - 1)
```

The function `lua_pushvalue` pushes on the stack a copy of the element at the given index.

The function `lua_rotate` is new in Lua 5.3. As the name implies, it rotates the stack elements from the given index to the top of the stack by `n` positions. A positive `n` rotates the elements in the direction of the top; a negative `n` rotates in the other direction. This is a quite versatile function, and two other API operations are defined as macros using it. One is `lua_remove`, which removes the element at the given index, shifting down the elements above this position to fill in the gap. Its definition is as follows:

```
#define lua_remove(L,idx)  \
          (lua_rotate(L, (idx), -1), lua_pop(L, 1))
```

That is, it rotates the stack by one position, moving the desired element to the top, and then pops that element. The other macro is `lua_insert`, which moves the top element into the given position, shifting up the elements above this position to open space:

```
#define lua_insert(L,idx)       lua_rotate(L, (idx), 1)
```

The function `lua_replace` pops a value and sets it as the value of the given index, without moving anything; finally, `lua_copy` copies the value at one index to another, leaving the original untouched.[5] Note that the following operations have no effect on a non-empty stack:

```
lua_settop(L, -1);  /* set top to its current value */
lua_insert(L, -1);  /* move top element to the top */
lua_copy(L, x, x);  /* copy an element to its own position */
lua_rotate(L, x, 0);  /* rotates by zero positions */
```

The program in Figure 27.3, “Example of stack manipulation” uses `stackDump` (defined in Figure 27.2, “Dumping the stack”) to illustrate these stack operations.

[5] The function `lua_copy` was introduced in Lua 5.2.

Figure 27.3. Example of stack manipulation

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"

static void stackDump (lua_State *L) {
  as in Figure 27.2, “Dumping the stack”
}

int main (void) {
  lua_State *L = luaL_newstate();

  lua_pushboolean(L, 1);
  lua_pushnumber(L, 10);
  lua_pushnil(L);
  lua_pushstring(L, "hello");

  stackDump(L);
    /* will print:  true  10  nil  'hello'  */

  lua_pushvalue(L, -4); stackDump(L);
    /* will print:  true  10  nil  'hello'  true  */

  lua_replace(L, 3); stackDump(L);
    /* will print:  true  10  true  'hello'  */

  lua_settop(L, 6); stackDump(L);
    /* will print:  true  10  true  'hello'  nil  nil  */

  lua_rotate(L, 3, 1); stackDump(L);
    /* will print:  true  10  nil  true  'hello'  nil  */

  lua_remove(L, -3); stackDump(L);
    /* will print:  true  10  nil  'hello'  nil */

  lua_settop(L, -5); stackDump(L);
    /* will print:  true  */

  lua_close(L);
  return 0;
}
```

Error Handling with the C API

All structures in Lua are dynamic: they grow as needed, and eventually shrink again when possible. This means that the possibility of a memory-allocation failure is pervasive in Lua. Almost any operation can face this eventuality. Moreover, many operations can raise other errors; for instance, an access to a global variable can trigger an `__index` metamethod and that metamethod may raise an error. Finally, operations that allocate memory eventually trigger the garbage collector, which may invoke finalizers, which can raise errors too. In short, the vast majority of functions in the Lua API can result in errors.

Instead of using error codes for each operation in its API, Lua uses exceptions to signal errors. Unlike C++ or Java, the C language does not offer an exception handling mechanism. To circumvent this difficulty, Lua uses the `setjmp` facility from C, which results in a mechanism somewhat similar to exception handling. Therefore, most API functions can raise an error (that is, call `longjmp`) instead of returning.

When we write library code (C functions to be called from Lua), the use of long jumps requires no extra work from our part, because Lua catches any error. When we write application code (C code that calls Lua), however, we must provide a way to catch those errors.

Error handling in application code

When our application calls functions in the Lua API, it is exposed to errors. As we just discussed, Lua usually signals these errors through long jumps. However, if there is no corresponding `setjmp`, the interpreter cannot make a long jump. In that case, any error in the API causes Lua to call a panic function and, if that function returns, exit the application. We can set our own panic function with `lua_atpanic`, but there is not much that it can do.

To properly handle errors in our application code, we must call our code through Lua, so that it sets an appropriate context to catch errors —that is, it runs the code in the context of a `setjmp`. In the same way that we can run Lua code in protected mode using `pcall`, we can run C code using `lua_pcall`. More specifically, we pack the code in a function and call that function through Lua, using `lua_pcall`. With this setting, our C code will run in protected mode. Even in case of memory-allocation failure, `lua_pcall` returns a proper error code, leaving the interpreter in a consistent state. The following fragment shows the idea:

```
static int foo (lua_State *L) {
  code to run in protected mode
  return 0;
}

int secure_foo (lua_State *L) {
  lua_pushcfunction(L, foo);  /* push 'foo' as a Lua function */
  return (lua_pcall(L, 0, 0, 0) == 0);
}
```

In this example, no matter what happens, a call to `secure_foo` will return a Boolean signaling the success of `foo`. In particular, note that the stack already has some preallocated slots and that `lua_pushcfunction` does not allocate memory, so it cannot raise any error. (The prototype of the function `foo` is a requirement of `lua_pushcfunction`, which creates a function in Lua representing a C function. We will cover the details about C functions in Lua in the section called “C Functions”.)

Error handling in library code

Lua is a *safe* language. This means that no matter what we write in Lua, no matter how wrong it is, we can always understand the behavior of a program in terms of Lua itself. Moreover, errors are detected and explained in terms of Lua, too. You can contrast that with C, where the behavior of many wrong programs can be explained only in terms of the underlying hardware (e.g., error positions are given as instruction addresses).

Whenever we add new C functions to Lua, we can break its safety. For instance, a function equivalent to the BASIC command `poke`, which stores an arbitrary byte at an arbitrary memory address, could cause all sorts of memory corruption. We must strive to ensure that our add-ons are safe to Lua and provide good error handling.

As we discussed earlier, C programs have to set their error handling through `lua_pcall`. When we write library functions for Lua, however, usually they do not need to handle errors. Errors raised by a library function will be caught either by a `pcall` in Lua or by a `lua_pcall` in the application code. So, whenever a function in a C library detects an error, it can simply call `lua_error` (or better yet `luaL_error`, which formats the error message and then calls `lua_error`). The function `lua_error` tidies any loose ends in the Lua system and jumps back to the protected call that originated that execution, passing along the error message.

Memory Allocation

The Lua core does not assume anything about how to allocate memory. It calls neither `malloc` nor `realloc` to allocate memory. Instead, it does all its memory allocation and deallocation through one single *allocation function*, which the user must provide when she creates a Lua state.

The function `luaL_newstate`, which we have been using to create states, is an auxiliary function that creates a Lua state with a default allocation function. This default allocation function uses the standard functions `malloc`–`realloc`–`free` from the C standard library, which are (or should be) good enough for most applications. However, it is quite easy to get full control over Lua allocation, by creating our state with the primitive `lua_newstate`:

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

This function takes two arguments: an allocation function and a user data. A state created in this way does all its allocation and deallocation by calling `f`; even the structure `lua_State` is allocated by `f`.

An allocation function must match the type `lua_Alloc`:

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

The first parameter is always the user data provided to `lua_newstate`; the second parameter is the address of the block being (re)allocated or released; the third parameter is the original block size; and the last parameter is the requested block size. If `ptr` is not `NULL`, Lua ensures that it was previously allocated with size `osize`. (When `ptr` is `NULL`, the previous size of the block was clearly zero, so Lua uses `osize` for some debug information.)

Lua uses `NULL` to represent a block of size zero. When `nsize` is zero, the allocation function must free the block pointed to by `ptr` and return `NULL`, which corresponds to a block of the required size (zero). When `ptr` is `NULL`, the function must allocate and return a block with the given size; if it cannot allocate the given block, it must return `NULL`. If `ptr` is `NULL` and `nsize` is zero, both rules apply: the net result is that the allocation function does nothing and returns `NULL`.

Finally, when `ptr` is non-`NULL` and `nsize` is non-zero, the allocation function should reallocate the block, like `realloc`, and return the new address (which may or may not be the same as the original). Again, in case of errors, it must return `NULL`. Lua assumes that the allocation function never fails when the new size is smaller than or equal to the old one. (Lua shrinks some structures during garbage collection, and it is unable to recover from errors there.)

The standard allocation function used by `luaL_newstate` has the following definition (extracted directly from the file `lauxlib.c`):

```
void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  (void)ud; (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

It assumes that `free(NULL)` does nothing and that `realloc(NULL, size)` is equivalent to `malloc(size)`. The ISO C standard mandates both behaviors.

We can recover the memory allocator of a Lua state by calling `lua_getallocf`:

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

If `ud` is not `NULL`, the function sets `*ud` with the value of the user data for this allocator. We can change the memory allocator of a Lua state by calling `lua_setallocf`:

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

Keep in mind that any new allocator will be responsible for freeing blocks that were allocated by the previous one. More often than not, the new function is a wrapper around the old one, for instance to trace allocations or to synchronize accesses to the heap.

Internally, Lua does not cache free memory blocks for reuse. It assumes that the allocation function does this caching; good allocators do. Lua also does not attempt to minimize fragmentation. Studies show that fragmentation is more the result of poor allocation strategies than of program behavior; good allocators do not create much fragmentation.

It is difficult to beat a well-implemented allocator, but sometimes you may try. For instance, Lua gives you the old size of any block that it frees or reallocates. Therefore, a specialized allocator does not need to keep information about the block size, reducing the memory overhead for each block.

Another situation where you can improve memory allocation is in multithreading systems. Such systems typically demand synchronization for their memory-allocation functions, as they use a global resource (the heap). However, the access to a Lua state must be synchronized too —or, better yet, restricted to one thread, as in our implementation of `lproc` in Chapter 33, *Threads and States*. So, if each Lua state allocates memory from a private pool, the allocator can avoid the costs of extra synchronization.

Exercises

Exercise 27.1: Compile and run the simple stand-alone interpreter (Figure 27.1, “A bare-bones stand-alone Lua interpreter”).

Exercise 27.2: Assume the stack is empty. What will be its contents after the following sequence of calls?

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_rotate(L, 1, -1);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```

Exercise 27.3: Use the function `stackDump` (Figure 27.2, “Dumping the stack”) to check your answer to the previous exercise.

Exercise 27.4: Write a library that allows a script to limit the total amount of memory used by its Lua state. It may offer a single function, `setlimit`, to set that limit.

The library should set its own allocation function. This function, before calling the original allocator, checks the total memory in use and returns `NULL` if the requested memory exceeds the limit.

(Hint: the library can use the user data of the allocation function to keep its state: the byte count, the current memory limit, etc.; remember to use the original user data when calling the original allocation function.)

Chapter 28. Extending Your Application

An important use of Lua is as a *configuration* language. In this chapter, we will illustrate how we can use Lua to configure a program, starting with a simple example and evolving it to perform increasingly complex tasks.

The Basics

As our first task, let us imagine a simple configuration scenario: our C program has a window and we want the user to be able to specify the initial window size. Clearly, for such a simple task, there are several options simpler than using Lua, like environment variables or files with name-value pairs. But even using a simple text file, we have to parse it somehow; so, we decide to use a Lua configuration file (that is, a plain text file that happens to be a Lua program). In its simplest form, this file can contain something like the following:

```
-- define window size
width = 200
height = 300
```

Now, we must use the Lua API to direct Lua to parse this file and then to get the values of the global variables `width` and `height`. The function `load`, in Figure 28.1, “Getting user information from a configuration file”, does this job.

Figure 28.1. Getting user information from a configuration file

```
int getglobint (lua_State *L, const char *var) {
  int isnum, result;
  lua_getglobal(L, var);
  result = (int)lua_tointegerx(L, -1, &isnum);
  if (!isnum)
    error(L, "'%s' should be a number\n", var);
  lua_pop(L, 1);   /* remove result from the stack */
  return result;
}

void load (lua_State *L, const char *fname, int *w, int *h) {
  if (luaL_loadfile(L, fname) || lua_pcall(L, 0, 0, 0))
    error(L, "cannot run config. file: %s", lua_tostring(L, -1));
  *w = getglobint(L, "width");
  *h = getglobint(L, "height");
}
```

top of stack

It assumes that we have already created a Lua state, following what we saw in the previous chapter. It calls `luaL_loadfile` to load the chunk from the file `fname`, and then calls `lua_pcall` to run the compiled chunk. In case of errors (e.g., a syntax error in our configuration file), these functions push the error message onto the stack and return a non-zero error code; our program then uses `lua_tostring` with index -1 to get the message from the top of the stack. (We defined the function `error` in the section called “A First Example”.)

After running the chunk, the program needs to get the values of the global variables. For that, it calls the auxiliary function `getglobint` (also in Figure 28.1, “Getting user information from a configuration file") twice. This function first calls `lua_getglobal`, whose single parameter (besides the omnipresent `lua_State`) is the variable name, to push the corresponding global value onto the stack. Next, `getglobint` uses `lua_tointegerx` to convert this value to an integer, ensuring that it has the correct type.

Is it worth using Lua for that task? As I said before, for such a simple task, a simple file with only two numbers in it would be easier to use than Lua. Even so, the use of Lua brings some advantages. First, Lua handles all syntax details for us; our configuration file can even have comments! Second, the user is already able to do some complex configurations with it. For instance, the script may prompt the user for some information, or it can query an environment variable to choose a proper size:

```
-- configuration file
if getenv("DISPLAY") == ":0.0" then
  width = 300; height = 300
else
  width = 200; height = 200
end
```

CONFIG FILE

Even in such simple configuration scenarios, it is hard to anticipate what users will want; but as long as the script defines the two variables, our C application works without changes.

A final reason for using Lua is that now it is easy to add new configuration facilities to our program; this ease fosters an attitude that results in programs that are more flexible.

Table Manipulation

Let us adopt that attitude: now, we want to configure a background color for the window, too. We will assume that the final color specification is composed of three numbers, where each number is a color component in RGB. Usually, in C, these numbers are integers in some range like *[0,255]*. In Lua, we will use the more natural range *[0,1]*.

A naive approach here is to ask the user to set each component in a different global variable:

```
-- configuration file
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

This approach has two drawbacks: it is too verbose (real programs may need dozens of different colors, for window background, window foreground, menu background, etc.); and there is no way to predefine common colors, so that, later, the user can simply write something like `background = WHITE`. To avoid these drawbacks, we will use a table to represent a color:

```
background = {red = 0.30, green = 0.10, blue = 0}
```

The use of tables gives more structure to the script; now it is easy for the user (or for the application) to predefine colors for later use in the configuration file:

```
BLUE = {red = 0, green = 0, blue = 1.0}
other color definitions

background = BLUE
```

To get these values in C, we can do as follows:

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
  error(L, "'background' is not a table");

red = getcolorfield(L, "red");
green = getcolorfield(L, "green");
blue = getcolorfield(L, "blue");
```

We first get the value of the global variable `background` and ensure that it is a table; then we use `getcolorfield` to get each color component.

Of course, the function `getcolorfield` is not part of the Lua API; we must define it. Again, we face the problem of polymorphism: there are potentially many versions of `getcolorfield` functions, varying the key type, value type, error handling, etc. The Lua API offers one function, `lua_gettable`, that works for all types. It takes the position of the table on the stack, pops the key from the stack, and pushes the corresponding value. Our private `getcolorfield`, defined in Figure 28.2, “A particular `getcolorfield` implementation”, assumes that the table is on the top of the stack; so, after pushing the key with `lua_pushstring`, the table will be at index -2. Before returning, `getcolorfield` pops the retrieved value from the stack, leaving the stack at the same level that it was before the call.

Figure 28.2. A particular `getcolorfield` implementation

```
#define MAX_COLOR       255

/* assume that table is on the top of the stack */
int getcolorfield (lua_State *L, const char *key) {
  int result, isnum;
  lua_pushstring(L, key);  /* push key */
  lua_gettable(L, -2);  /* get background[key] */
  result = (int)(lua_tonumberx(L, -1, &isnum) * MAX_COLOR);
  if (!isnum)
    error(L, "invalid component '%s' in color", key);
  lua_pop(L, 1);  /* remove number */
  return result;
}
```

We will extend our example a little further and introduce color names for the user. The user can still use color tables, but she can also use predefined names for the more common colors. To implement this feature, we need a color table in our C application:

```
struct ColorTable {
  char *name;
  unsigned char red, green, blue;
} colortable[] = {
  {"WHITE",   MAX_COLOR, MAX_COLOR, MAX_COLOR},
  {"RED",     MAX_COLOR,         0,         0},
  {"GREEN",           0, MAX_COLOR,         0},
  {"BLUE",            0,         0, MAX_COLOR},
  other colors
  {NULL, 0, 0, 0}  /* sentinel */
};
```

Our implementation will create global variables with the color names and initialize these variables using color tables. The result is the same as if the user had the following lines in her script:

```
WHITE = {red = 1.0, green = 1.0, blue = 1.0}
RED   = {red = 1.0, green = 0,   blue = 0}
other colors
```

To set the table fields, we define an auxiliary function, setcolorfield; it pushes the index and the field value on the stack, and then calls lua_settable:

```
/* assume that table is on top */
void setcolorfield (lua_State *L, const char *index, int value) {
  lua_pushstring(L, index); /* key */
  lua_pushnumber(L, (double)value / MAX_COLOR);  /* value */
  lua_settable(L, -3);
}
```

Like other API functions, lua_settable works for many different types, so it gets all its operands from the stack. It takes the table index as an argument and pops the key and the value. The function setcolorfield assumes that before the call the table is on the top of the stack (index -1); after pushing the index and the value, the table will be at index -3.

The next function, setcolor, defines a single color. It creates a table, sets the appropriate fields, and assigns this table to the corresponding global variable:

```
void setcolor (lua_State *L, struct ColorTable *ct) {
  lua_newtable(L);                 /* creates a table */
  setcolorfield(L, "red", ct->red);
  setcolorfield(L, "green", ct->green);
  setcolorfield(L, "blue", ct->blue);
  lua_setglobal(L, ct->name);      /* 'name' = table */
}
```

The function lua_newtable creates an empty table and pushes it on the stack; the three calls to setcolorfield set the table fields; finally, lua_setglobal pops the table and sets it as the value of the global with the given name.

With these previous functions, the following loop will register all colors for the configuration script:

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

Remember that the application must execute this loop before running the script.

Figure 28.3, “Colors as strings or tables” shows another option for implementing named colors.

Figure 28.3. Colors as strings or tables

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) {   /* value is a string? */
  const char *name = lua_tostring(L, -1);  /* get string */
  int i;   /* search the color table */
  for (i = 0; colortable[i].name != NULL; i++) {
    if (strcmp(colorname, colortable[i].name) == 0)
      break;
  }
  if (colortable[i].name == NULL)  /* string not found? */
    error(L, "invalid color name (%s)", colorname);
  else {  /* use colortable[i] */
    red = colortable[i].red;
    green = colortable[i].green;
    blue = colortable[i].blue;
  }
} else if (lua_istable(L, -1)) {
  red = getcolorfield(L, "red");
  green = getcolorfield(L, "green");
  blue = getcolorfield(L, "blue");
} else
    error(L, "invalid value for 'background'");
```

Instead of global variables, the user can denote color names with strings, writing her settings as `background = "BLUE"`. Therefore, `background` can be either a table or a string. With this design, the application does not need to do anything before running the user's script. Instead, it needs more work to get a color. When it gets the value of the variable `background`, it must test whether the value is a string, and then look up the string in the color table.

What is the best option? In C programs, the use of strings to denote options is not a good practice, because the compiler cannot detect misspellings. In Lua, however, the error message for a misspelt color will probably be seen by the author of the configuration “program”. The distinction between programmer and user is blurred, and so the difference between a compilation error and a run-time error is blurred, too.

With strings, the value of `background` would be the misspelled string; hence, the application can add this information to the error message. The application can also compare strings regardless of case, so that a user can write `"white"`, `"WHITE"`, or even `"White"`. Moreover, if the user script is small and there are many colors, it may be inefficient to register hundreds of colors (and to create hundreds of tables and global variables) when the user needs only a few. With strings, we avoid this overhead.

Some short cuts

Although the C API strives for simplicity, Lua is not radical. So, the API offers short cuts for several common operations. Let us see some of them.

Because indexing a table with a string key is so common, Lua has a specialized version of `lua_gettable` for this case: `lua_getfield`. Using this function, we can rewrite the two lines

```
lua_pushstring(L, key);
lua_gettable(L, -2);  /* get background[key] */
```

in `getcolorfield` as

```
lua_getfield(L, -1, key);  /* get background[key] */
```

(As we do not push the string onto the stack, the table index is still -1 when we call lua_getfield.)

Because it is common to check the type of a value returned by lua_gettable, in Lua 5.3 this function (and similar ones like lua_getfield) now returns the type of its result. Therefore, we can simplify further the access and the check in getcolorfield:

```
    if (lua_getfield(L, -1, key) != LUA_TNUMBER)
      error(L, "invalid component in background color");
```

As you might expect, Lua offers also a specialized version of lua_settable for string keys, called lua_setfield. Using this function, we can rewrite our previous definition for setcolorfield as follows:

```
void setcolorfield (lua_State *L, const char *index, int value) {
  lua_pushnumber(L, (double)value / MAX_COLOR);
  lua_setfield(L, -2, index);
}
```

As a small optimization, we can also replace our use of lua_newtable in the function setcolor. Lua offers another function, lua_createtable, where we create a table and pre-allocate space for entries. Lua declares these functions like this:

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

#define lua_newtable(L)      lua_createtable(L, 0, 0)
```

The parameter narr is the expected number of elements in the sequence part of the table (that is, entries with sequential integer indices), and nrec is the expected number of other elements. In setcolor, we could write lua_createtable(L, 0, 3) as a hint that the table will get three entries. (Lua code does a similar optimization when we write a constructor.)

Calling Lua Functions

A great strength of Lua is that a configuration file can define functions to be called by the application. For instance, we can write in C an application to plot the graph of a function and define in Lua the function to be plotted.

The API protocol to call a function is simple: first, we push the function to be called; second, we push the arguments to the call; then we use lua_pcall to do the actual call; finally, we get the results from the stack.

As an example, let us assume that our configuration file has a function like this:

```
function f (x, y)
  return (x^2 * math.sin(y)) / (1 - x)
end
```

We want to evaluate, in C, z = f(x, y) for given x and y. Assuming that we have already opened the Lua library and run the configuration file, the function f in Figure 28.4, “Calling a Lua function from C” evaluates that code.

Figure 28.4. Calling a Lua function from C

```
/* call a function 'f' defined in Lua */
double f (lua_State *L, double x, double y) {
  int isnum;
  double z;

  /* push functions and arguments */
  lua_getglobal(L, "f");  /* function to be called */
  lua_pushnumber(L, x);   /* push 1st argument */
  lua_pushnumber(L, y);   /* push 2nd argument */

  /* do the call (2 arguments, 1 result) */
  if (lua_pcall(L, 2, 1, 0) != LUA_OK)
    error(L, "error running function 'f': %s",
             lua_tostring(L, -1));

  /* retrieve result */
  z = lua_tonumberx(L, -1, &isnum);
  if (!isnum)
    error(L, "function 'f' should return a number");
  lua_pop(L, 1);  /* pop returned value */
  return z;
}
```

The second and third arguments to `lua_pcall` are the number of arguments we are passing and the number of results we want. The fourth argument indicates a message-handling function; we will discuss it in a moment. As in a Lua assignment, `lua_pcall` adjusts the actual number of results to what we have asked for, pushing nils or discarding extra values as needed. Before pushing the results, `lua_pcall` removes from the stack the function and its arguments. When a function returns multiple results, the first result is pushed first; for instance, if there are three results, the first one will be at index -3 and the last at index -1.

If there is any error while `lua_pcall` is running, `lua_pcall` returns an error code; moreover, it pushes the error message on the stack (but still pops the function and its arguments). Before pushing the message, however, `lua_pcall` calls the message handler function, if there is one. To specify a message handler function, we use the last argument of `lua_pcall`. Zero means no message handler function; that is, the final error message is the original message. Otherwise, this argument should be the index on the stack where the message handler function is located. In such cases, we should push the handler on the stack somewhere below the function to be called.

For normal errors, `lua_pcall` returns the error code `LUA_ERRRUN`. Two special kinds of errors deserve different codes, because they never run the message handler. The first kind is a memory allocation error. For such errors, `lua_pcall` returns `LUA_ERRMEM`. The second kind is an error while Lua is running the message handler itself. In this case, it is of little use to call the handler again, so `lua_pcall` returns immediately with a code `LUA_ERRERR`. Since version 5.2, Lua differentiates a third kind of error: when a finalizer raises an error, `lua_pcall` returns the code `LUA_ERRGCMM` (*error in a GC metamethod*). This code indicates that the error is not directly related to the call itself.

A Generic Call Function

As a more advanced example, we will build a wrapper for calling Lua functions, using the `stdarg` facility in C. Our wrapper function, let us call it `call_va`, takes the name of a global function to be called, a string describing the types of the arguments and results, then the list of arguments, and finally a list of pointers to variables to store the results; it handles all the details of the API. With this function, we could write our example in Figure 28.4, “Calling a Lua function from C” simply like this:

```
call_va(L, "f", "dd>d", x, y, &z);
```

The string `"dd>d"` means “two arguments of type double, one result of type double”. This descriptor can use the letters `d` for double, `i` for integer, and `s` for strings; a `>` separates arguments from the results. If the function has no results, the `>` is optional.

Figure 28.5, “A generic call function” shows the implementation of `call_va`.

Figure 28.5. A generic call function

```
#include <stdarg.h>

void call_va (lua_State *L, const char *func,
                            const char *sig, ...) {
  va_list vl;
  int narg, nres;  /* number of arguments and results */

  va_start(vl, sig);
  lua_getglobal(L, func);  /* push function */

  push and count arguments (Figure 28.6, “Pushing arguments for the generic

  nres = strlen(sig);  /* number of expected results */

  if (lua_pcall(L, narg, nres, 0) != 0)  /* do the call */
    error(L, "error calling '%s': %s", func,
                                   lua_tostring(L, -1));

  retrieve results (Figure 28.7, “Retrieving results for the generic call fu

  va_end(vl);
}
```

Despite its generality, this function follows the same steps of our first example: it pushes the function, pushes the arguments (Figure 28.6, “Pushing arguments for the generic call function”), does the call, and gets the results (Figure 28.7, “Retrieving results for the generic call function”).

Figure 28.6. Pushing arguments for the generic call function

```
for (narg = 0; *sig; narg++) {  /* repeat for each argument */

  /* check stack space */
  luaL_checkstack(L, 1, "too many arguments");

  switch (*sig++) {

    case 'd':  /* double argument */
      lua_pushnumber(L, va_arg(vl, double));
      break;

    case 'i':  /* int argument */
      lua_pushinteger(L, va_arg(vl, int));
      break;

    case 's':  /* string argument */
      lua_pushstring(L, va_arg(vl, char *));
      break;

    case '>':  /* end of arguments */
      goto endargs;  /* break the loop */

    default:
      error(L, "invalid option (%c)", *(sig - 1));
  }

}
endargs:
```

Figure 28.7. Retrieving results for the generic call function

```
nres = -nres;  /* stack index of first result */
while (*sig) {  /* repeat for each result */
  switch (*sig++) {

    case 'd': {  /* double result */
      int isnum;
      double n = lua_tonumberx(L, nres, &isnum);
      if (!isnum)
        error(L, "wrong result type");
      *va_arg(vl, double *) = n;
      break;
    }

    case 'i': {  /* int result */
      int isnum;
      int n = lua_tointegerx(L, nres, &isnum);
      if (!isnum)
        error(L, "wrong result type");
      *va_arg(vl, int *) = n;
      break;
    }

    case 's': {  /* string result */
      const char *s = lua_tostring(L, nres);
      if (s == NULL)
        error(L, "wrong result type");
      *va_arg(vl, const char **) = s;
      break;
    }

    default:
      error(L, "invalid option (%c)", *(sig - 1));
  }
  nres++;
}
```

Most of its code is straightforward, but there are some subtleties. First, it does not need to check whether `func` is a function: `lua_pcall` will trigger that error. Second, because it pushes an arbitrary number of arguments, it must ensure that there is enough stack space. Third, because the function can return strings, `call_va` cannot pop the results from the stack. It is up to the caller to pop them, after it finishes using any string results (or after copying them to appropriate buffers).

Exercises

Exercise 28.1: Write a C program that reads a Lua file defining a function `f` from numbers to numbers and plots that function. (You do not need to do anything fancy; the program can plot the results printing ASCII asterisks as we did in the section called “Compilation”.)

Exercise 28.2: Modify the function `call_va` (Figure 28.5, “A generic call function”) to handle Boolean values.

Exercise 28.3: Let us suppose a program that needs to monitor several weather stations. Internally, it uses a four-byte string to represent each station, and there is a configuration file to map each string to the actual URL of the corresponding station. A Lua configuration file could do this mapping in several ways:

- a bunch of global variables, one for each station;
- a table mapping string codes to URLs;
- a function mapping string codes to URLs.

Discuss the pros and cons of each option, considering things like the total number of stations, the regularity of the URLs (e.g., there may be a formation rule from codes to URLs), the kind of users, etc.

Chapter 29. Calling C from Lua

When we say that Lua can call C functions, this does not mean that Lua can call any C function.[1] As we saw in the previous chapter, when C calls a Lua function, it must follow a simple protocol to pass the arguments and to get the results. Similarly, for Lua to call a C function, the C function must follow a protocol to get its arguments and to return its results. Moreover, for Lua to call a C function, we must register the function, that is, we must give its address to Lua in an appropriate way.

When Lua calls a C function, it uses the same kind of stack that C uses to call Lua. The C function gets its arguments from the stack and pushes the results on the stack.

An important point here is that the stack is not a global structure; each function has its own private local stack. When Lua calls a C function, the first argument will always be at index 1 of this local stack. Even when a C function calls Lua code that calls the same (or another) C function again, each of these invocations sees only its own private stack, with its first argument at index 1.

C Functions

As a first example, let us see how to implement a simplified version of a function that returns the sine of a given number:

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1);  /* get argument */
  lua_pushnumber(L, sin(d));  /* push result */
  return 1;  /* number of results */
}
```

Any function registered with Lua must have this same prototype, defined in `lua.h` as `lua_CFunction`:

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

prototype

From the point of view of C, a C function gets as its single argument the Lua state and returns an integer with the number of values it is returning on the stack. Therefore, the function does not need to clear the stack before pushing its results. After it returns, Lua automatically saves its results and clears its entire stack.

Before we can call this function from Lua, we must register it. We do this bit of magic with `lua_pushcfunction`: it gets a pointer to a C function and creates a value of type `"function"` that represents this function inside Lua. Once registered, a C function behaves like any other function inside Lua.

A quick-and-dirty way to test our function `l_sin` is to put its code directly into our basic interpreter (Figure 27.1, “A bare-bones stand-alone Lua interpreter”) and add the following lines right after the call to `luaL_openlibs`:

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

The first line pushes a value of type function; the second line assigns it to the global variable `mysin`. After these modifications, we can use the new function `mysin` in our Lua scripts. In the next section, we will discuss better ways to link new C functions with Lua. Here, we will explore how to write better C functions.

[1] There are packages that allow Lua to call any C function, but they are neither as portable as Lua nor safe.

For a more professional sine function, we must check the type of its argument. The auxiliary library helps us with this task. The function luaL_checknumber checks whether a given argument is a number: in case of error, it throws an informative error message; otherwise, it returns the number. The modification to our function is minimal:

```
static int l_sin (lua_State *L) {
  double d = luaL_checknumber(L, 1);
  lua_pushnumber(L, sin(d));
  return 1;  /* number of results */
}
```

With the above definition, if you call mysin('a'), you get an error like this one:

```
bad argument #1 to 'mysin' (number expected, got string)
```

The function luaL_checknumber automatically fills the message with the argument number (#1), the function name ("mysin"), the expected parameter type (number), and the actual parameter type (string).

As a more complex example, let us write a function that returns the contents of a given directory. Lua does not provide this function in its standard libraries, because ISO C does not offer functions for this job. Here, we will assume that we have a POSIX compliant system. Our function—we will call it dir in Lua, l_dir in C—gets as argument a string with the directory path and returns a list with the directory entries. For instance, a call like dir("/home/lua") may return the table {".", "..", "src", "bin", "lib"}. The complete code for this function is in Figure 29.1, “A function to read a directory”.

Figure 29.1. A function to read a directory

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

#include "lua.h"
#include "lauxlib.h"

static int l_dir (lua_State *L) {
  DIR *dir;
  struct dirent *entry;
  int i;
  const char *path = luaL_checkstring(L, 1);

  /* open directory */
  dir = opendir(path);
  if (dir == NULL) {  /* error opening the directory? */
    lua_pushnil(L);  /* return nil... */
    lua_pushstring(L, strerror(errno));  /* and error message */
    return 2;  /* number of results */
  }

  /* create result table */
  lua_newtable(L);
  i = 1;
  while ((entry = readdir(dir)) != NULL) {  /* for each entry */
    lua_pushinteger(L, i++);  /* push key */
    lua_pushstring(L, entry->d_name);  /* push value */
    lua_settable(L, -3);    /* table[i] = entry name */
  }

  closedir(dir);
  return 1;  /* table is already on top */
}
```

It starts getting the directory path with `luaL_checkstring`, which is the equivalent of `luaL_checknumber` for strings. Then it opens this directory with `opendir`. In case it cannot open the directory, the function returns nil plus an error message that it gets with `strerror`. After opening the directory, the function creates a new table and populates it with the directory entries. (Each time we call `readdir`, it returns a next entry.) Finally, it closes the directory and returns 1, in C, meaning that it is returning the value on top of its stack to Lua. (Remember that `lua_settable` pops the key and the value from the stack. Therefore, after the loop, the element on the top of the stack is the result table.)

In some conditions, this implementation of `l_dir` may cause a memory leak. Three of the Lua functions that it calls can fail due to insufficient memory: `lua_newtable`, `lua_pushstring`, and `lua_settable`. If any of these functions fails, it will raise an error and interrupt `l_dir`, which therefore will not call `closedir`. In Chapter 32, *Managing Resources*, we will see an alternative implementation for a directory function that corrects this problem.

Continuations

Through `lua_pcall` and `lua_call`, a C function called from Lua can call Lua back. Several functions in the standard library do that: `table.sort` can call an order function; `string.gsub` can call a replacement function; pcall and xpcall call functions in protected mode. If we remember that the main Lua code was itself called from C (the host program), we have a call sequence like C (host) calls Lua (script) that calls C (library) that calls Lua (callback).

Usually, Lua handles these sequences of calls without problems; after all, this integration with C is a hallmark of the language. There is one situation, however, where this interlacement can cause difficulties: coroutines.

Each coroutine in Lua has its own stack, which keeps information about the pending calls of the coroutine. Specifically, the stack stores the return address, the parameters, and the local variables of each call. For calls to Lua functions, the interpreter needs only this stack, which we call the *soft stack*. For calls to C functions, however, the interpreter must use the C stack, too. After all, the return address and the local variables of a C function live in the C stack.

It is easy for the interpreter to have multiple soft stacks, but the runtime of ISO C has only one internal stack. Therefore, coroutines in Lua cannot suspend the execution of a C function: if there is a C function in the call path from a resume to its respective yield, Lua cannot save the state of that C function to restore it in the next resume. Consider the next example, in Lua 5.1:

```
co = coroutine.wrap(function ()
                      print(pcall(coroutine.yield))
                    end)
co()
  --> false    attempt to yield across metamethod/C-call boundary
```

The function pcall is a C function; therefore, Lua 5.1 cannot suspend it, because there is no way in ISO C to suspend a C function and resume it later.

Lua 5.2 and later versions ameliorated that difficulty with *continuations*. Lua 5.2 implements yields using long jumps, in the same way that it implements errors. A long jump simply throws away any information about C functions in the C stack, so it is impossible to resume those functions. However, a C function foo can specify a continuation function foo_k, which is another C function to be called when it is time to resume foo. That is, when the interpreter detects that it should resume foo, but that a long jump threw away the entry for foo in the C stack, it calls foo_k instead.

To make things a little more concrete, let us see the implementation of pcall as an example. In Lua 5.1, this function had the following code:

```
static int luaB_pcall (lua_State *L) {
  int status;
  luaL_checkany(L, 1);  /* at least one parameter */
  status = lua_pcall(L, lua_gettop(L) - 1, LUA_MULTRET, 0);
  lua_pushboolean(L, (status == LUA_OK));  /* status */
  lua_insert(L, 1);  /* status is first result */
  return lua_gettop(L);  /* return status + all results */
}
```

If the function being called through lua_pcall yielded, it would be impossible to resume luaB_pcall later. Therefore, the interpreter raised an error whenever we attempted to yield inside a protected call. Lua 5.3 implements pcall roughly like in Figure 29.2, “Implementation of pcall with continuations”.[2]

[2] The API for continuations in Lua 5.2 is a little different. Check the reference manual for details.

Figure 29.2. Implementation of `pcall` with continuations

```
static int finishpcall (lua_State *L, int status, intptr_t ctx) {
  (void)ctx;   /* unused parameter */
  status = (status != LUA_OK && status != LUA_YIELD);
  lua_pushboolean(L, (status == 0));  /* status */
  lua_insert(L, 1);  /* status is first result */
  return lua_gettop(L);  /* return status + all results */
}

static int luaB_pcall (lua_State *L) {
  int status;
  luaL_checkany(L, 1);
  status = lua_pcallk(L, lua_gettop(L) - 1, LUA_MULTRET, 0,
                      0, finishpcall);
  return finishpcall(L, status, 0);
}
```

There are three important differences from the Lua 5.1 version: first, the new version replaces the call to `lua_pcall` by a call to `lua_pcallk`; second, it puts everything done after that call in a new auxiliary function `finishpcall`; third, the status returned by `lua_pcallk` can be `LUA_YIELD`, besides `LUA_OK` or an error.

If there are no yields, `lua_pcallk` works exactly like `lua_pcall`. If there is a yield, however, then things are quite different. If a function called by the original `lua_pcall` tries to yield, Lua 5.3 raises an error, like Lua 5.1. But when a function called by the new `lua_pcallk` yields, there is no error: Lua does a long jump and discards the entry for `luaB_pcall` from the C stack, but keeps in the soft stack of the coroutine a reference to the *continuation function* given to `lua_pcallk` (`finishpcall`, in our example). Later, when the interpreter detects that it should return to `luaB_pcall` (which is impossible), it instead calls the continuation function.

The continuation function `finishpcall` can also be called when there is an error. Unlike the original `luaB_pcall`, `finishpcall` cannot get the value returned by `lua_pcallk`. So, it gets this value as an extra parameter, `status`. When there are no errors, `status` is `LUA_YIELD` instead of `LUA_OK`, so that the continuation function can check how it is being called. In case of errors, `status` is the original error code.

Besides the status of the call, the continuation function also receives a *context*. The fifth parameter to `lua_pcallk` is an arbitrary integer that is passed as the last parameter to the continuation function. (The type of this parameter, `intptr_t`, allows pointers to be passed as context, too.) This value allows the original function to pass some arbitrary information directly to its continuation. (Our example does not use this facility.)

The continuation system of Lua 5.3 is an ingenious mechanism to support yields, but it is not a panacea. Some C functions would need to pass too much context to their continuations. Examples include `table.sort`, which uses the C stack for recursion, and `string.gsub`, which must keep track of captures and a buffer for its partial result. Although it is possible to rewrite them in a “yieldable” way, the gains do not seem to be worth the extra complexity and performance losses.

C Modules

A Lua module is a chunk that defines several Lua functions and stores them in appropriate places, typically as entries in a table. A C module for Lua mimics this behavior. Besides the definition of its C functions, it must also define a special function that plays the role of the main chunk in a Lua library. This function should register all C functions of the module and store them in appropriate places, again typically as entries in a table. Like a Lua main chunk, it should also initialize anything else that needs initialization in the module.

Lua perceives C functions through this registration process. Once a C function is represented and stored in Lua, Lua calls it through a direct reference to its address (which is what we give to Lua when we register a function). In other words, Lua does not depend on a function name, package location, or visibility rules to call a function, once it is registered. Typically, a C module has one single public (extern) function, which is the function that opens the library. All other functions can be private, declared as `static` in C.

When we extend Lua with C functions, it is a good idea to design our code as a C module, even when we want to register only one C function: sooner or later (usually sooner) we will need other functions. As usual, the auxiliary library offers a helper function for this job. The macro `luaL_newlib` takes an array of C functions with their respective names and registers all of them inside a new table. As an example, suppose we want to create a library with the function `l_dir` that we defined earlier. First, we must define the library functions:

```
static int l_dir (lua_State *L) {
   as before
}
```

Next, we declare an array with all functions in the module with their respective names. This array has elements of type `luaL_Reg`, which is a structure with two fields: a function name (a string) and a function pointer.

```
static const struct luaL_Reg mylib [] = {
  {"dir", l_dir},
  {NULL, NULL}  /* sentinel */
};
```

In our example, there is only one function (`l_dir`) to declare. The last pair in the array is always `{NULL, NULL}`, to mark its end. Finally, we declare a main function, using `luaL_newlib`:

```
int luaopen_mylib (lua_State *L) {
  luaL_newlib(L, mylib);
  return 1;
}
```

The call to `luaL_newlib` creates a new table and fills it with the pairs name–function specified by the array `mylib`. When it returns, `luaL_newlib` leaves on the stack the new table wherein it opened the library. The function `luaopen_mylib` then returns 1 to return this table to Lua.

After finishing the library, we must link it to the interpreter. The most convenient way to do it is with the dynamic linking facility, if your Lua interpreter supports this facility. In this case, you must create a dynamic library with your code (`mylib.dll` in Windows, `mylib.so` in Linux-like systems) and put it somewhere in the C path. After these steps, you can load your library directly from Lua, with `require`:

```
local mylib = require "mylib"
```

This call links the dynamic library `mylib` with Lua, finds the function `luaopen_mylib`, registers it as a C function, and calls it, opening the module. (This behavior explains why `luaopen_mylib` must have the same prototype as any other C function.)

libname
dynamic lib
*

The dynamic linker must know the name of the function `luaopen_mylib` in order to find it. It will always look for `luaopen_` concatenated with the name of the module. Therefore, if our module is called mylib, that function should be called luaopen_mylib. (We discussed the details of this function name in Chapter 17, *Modules and Packages*.)

STATIC LINKING

If your interpreter does not support dynamic linking, then you have to recompile Lua with your new library. Besides this recompilation, you need some way of telling the stand-alone interpreter that it should open this library when it opens a new state. A simple way to do this is to add luaopen_mylib into the list of standard libraries to be opened by luaL_openlibs, in the file linit.c.

Exercises

Exercise 29.1: Write a variadic summation function, in C, that computes the sum of its variable number of numeric arguments:

```
print(summation())                   -->   0
print(summation(2.3, 5.4))           -->   7.7
print(summation(2.3, 5.4, -34))      --> -26.3
print(summation(2.3, 5.4, {}))
  --> stdin:1: bad argument #3 to 'summation'
                 (number expected, got table)
```

Exercise 29.2: Implement a function equivalent to table.pack, from the standard library.

Exercise 29.3: Write a function that takes any number of parameters and returns them in reverse order.

```
print(reverse(1, "hello", 20))   --> 20    hello    1
```

Exercise 29.4: Write a function foreach that takes a table and a function and calls that function for each key–value pair in the table.

```
foreach({x = 10, y = 20}, print)
  --> x    10
  --> y    20
```

(Hint: check the function lua_next in the Lua manual.)

Exercise 29.5: Rewrite the function foreach, from the previous exercise, so that the function being called can yield.

Exercise 29.6: Create a C module with all functions from the previous exercises.

Chapter 30. Techniques for Writing C Functions

Both the official API and the auxiliary library provide several mechanisms to help writing C functions. In this chapter, we cover the mechanisms for array manipulation, string manipulation, and storing Lua values in C.

Array Manipulation

An “array”, in Lua, is just a table used in a specific way. We can manipulate arrays using the same generic functions we use to manipulate tables, namely `lua_settable` and `lua_gettable`. However, the API provides special functions to access and update tables with integer keys:

```
void lua_geti (lua_State *L, int index, int key);
void lua_seti (lua_State *L, int index, int key);
```

Lua versions prior to 5.3 offered only raw versions of these functions, `lua_rawgeti` and `lua_rawseti`. They are similar to `lua_geti` and `lua_seti`, but do raw accesses (that is, without invoking metamethods). When the difference in unimportant (e.g., the table has no metamethods), the raw versions can be slightly faster.

The description of `lua_geti` and `lua_seti` is a little confusing, as it involves two indices: `index` refers to where the table is on the stack; `key` refers to where the element is in the table. The call `lua_geti(L, t, key)` is equivalent to the following sequence when `t` is positive (otherwise, we must compensate for the new item on the stack):

```
lua_pushnumber(L, key);
lua_gettable(L, t);
```

The call `lua_seti(L, t, key)` (again for `t` positive) is equivalent to this sequence:

```
lua_pushnumber(L, key);
lua_insert(L, -2);  /* put 'key' below previous value */
lua_settable(L, t);
```

As a concrete example of the use of these functions, Figure 30.1, “The function `map` in C” implements the function map: it applies a given function to all elements of an array, replacing each element by the result of the call.

Figure 30.1. The function map in C

```
int l_map (lua_State *L) {
  int i, n;

  /* 1st argument must be a table (t) */
  luaL_checktype(L, 1, LUA_TTABLE);

  /* 2nd argument must be a function (f) */
  luaL_checktype(L, 2, LUA_TFUNCTION);

  n = luaL_len(L, 1);  /* get size of table */

  for (i = 1; i <= n; i++) {
    lua_pushvalue(L, 2);   /* push f */
    lua_geti(L, 1, i);  /* push t[i] */
    lua_call(L, 1, 1);     /* call f(t[i]) */
    lua_seti(L, 1, i);  /* t[i] = result */
  }

  return 0;  /* no results */
}
```

This example also introduces three new functions: `luaL_checktype`, `luaL_len`, and `lua_call`.

The function `luaL_checktype` (from `lauxlib.h`) ensures that a given argument has a given type; otherwise, it raises an error.

The primitive `lua_len` (not used in the example) is equivalent to the length operator. Because of metamethods, this operator may result in any kind of object, not only numbers; therefore, `lua_len` returns its result on the stack. The function `luaL_len` (the one used in the example, from the auxiliary library) returns the length as an integer, raising an error if the coercion is not possible.

The function `lua_call` does an unprotected call. It is similar to `lua_pcall`, but it propagates errors, instead of returning an error code. When we are writing the main code in an application, we should not use `lua_call`, because we want to catch any errors. When we are writing functions, however, it is usually a good idea to use `lua_call`; if there is an error, just leave it to someone who cares about it.

String Manipulation

When a C function receives a string argument from Lua, there are only two rules that it must observe: not to pop the string from the stack while using it and never to modify the string.

Things get more demanding when a C function needs to create a string to return to Lua. Now, it is up to the C code to take care of buffer allocation/deallocation, buffer overflows, and other tasks that are difficult in C. So, the Lua API provides some functions to help with these tasks.

The standard API provides support for two of the most basic string operations: substring extraction and string concatenation. To extract a substring, remember that the basic operation `lua_pushlstring` gets the string length as an extra argument. Therefore, if we want to pass to Lua a substring of a string `s` ranging from position `i` to `j` (inclusive), all we have to do is this:

```
lua_pushlstring(L, s + i, j - i + 1);
```

As an example, suppose we want a function that splits a string according to a given separator (a single character) and returns a table with the substrings. For instance, the call `split("hi:ho:there", ":")` should return the table `{"hi", "ho", "there"}`. Figure 30.2, "Splitting a string" presents a simple implementation for this function.

Figure 30.2. Splitting a string

```
static int l_split (lua_State *L) {
  const char *s = luaL_checkstring(L, 1);      /* subject */
  const char *sep = luaL_checkstring(L, 2);  /* separator */
  const char *e;
  int i = 1;

  lua_newtable(L);  /* result table */

  /* repeat for each separator */
  while ((e = strchr(s, *sep)) != NULL) {
    lua_pushlstring(L, s, e - s);  /* push substring */
    lua_rawseti(L, -2, i++);   /* insert it in table */
    s = e + 1;  /* skip separator */
  }

  /* insert last substring */
  lua_pushstring(L, s);
  lua_rawseti(L, -2, i);

  return 1;  /* return the table */
}
```

It uses no buffers and can handle arbitrarily long strings: Lua takes care of all the memory allocation. (As we created the table, we know it has no metatable; so, we can manipulate it with the raw operations.)

To concatenate strings, Lua provides a specific function, called `lua_concat`. It is equivalent to the concatenation operator (`..`) in Lua: it converts numbers to strings and triggers metamethods when necessary. Moreover, it can concatenate more than two strings at once. The call `lua_concat(L, n)` will concatenate (and pop) the top-most `n` values on the stack and push the result.

Another helpful function is `lua_pushfstring`:

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

It is somewhat similar to the C function `sprintf`, in that it creates a string according to a format string and some extra arguments. Unlike `sprintf`, however, we do not need to provide a buffer. Lua dynamically creates the string for us, as large as it needs to be. The function pushes the resulting string on the stack and returns a pointer to it. This function accepts the following directives:

`%s`	inserts a zero-terminated string
`%d`	inserts an `int`
`%f`	inserts a Lua float
`%p`	inserts a pointer
`%I`	inserts a Lua integer
`%c`	inserts an `int` as a one-byte character

%U	inserts an int as a UTF-8 byte sequence
%%	inserts a percent sign

It accepts no modifiers, such as width or precision.[1]

Both lua_concat and lua_pushfstring are useful when we want to concatenate only a few strings. However, if we need to concatenate many strings (or characters) together, a one-by-one approach can be quite inefficient, as we saw in the section called “String Buffers”. Instead, we can use the *buffer facility* provided by the auxiliary library.

In its simpler usage, the buffer facility works with two functions: one gives us a buffer of any size where we can compose our string; the other converts the contents of the buffer into a Lua string.[2] Figure 30.3, “The function string.upper” illustrates those functions with the implementation of string.upper, right from the source file lstrlib.c.

Figure 30.3. The function string.upper

```
static int str_upper (lua_State *L) {
  size_t l;
  size_t i;
  luaL_Buffer b;
  const char *s = luaL_checklstring(L, 1, &l);
  char *p = luaL_buffinitsize(L, &b, l);
  for (i = 0; i < l; i++)
    p[i] = toupper(uchar(s[i]));
  luaL_pushresultsize(&b, l);
  return 1;
}
```

The first step for using a buffer from the auxiliary library is to declare a variable with type luaL_Buffer. The next step is to call luaL_buffinitsize to get a pointer for a buffer with the given size; we can then use this buffer freely to create our string. The last step is to call luaL_pushresultsize to convert the buffer contents into a new Lua string and push that sting onto the stack. The size in this second call is the final size of the string. Often, as in our example, this size is equal to the size of the buffer, but it can be smaller. If we do not know the exact size of the resulting string, but have an upper bound, we can conservatively allocate a larger size.

Note that luaL_pushresultsize does not get a Lua state as its first argument. After the initialization, a buffer keeps a reference to the state, so we do not need to pass it when calling other functions that manipulate buffers.

We can also use the auxlib buffers by adding content to them piecemeal, without knowing an upper bound on the size of the result. The auxiliary library offers several functions to add things to a buffer: luaL_addvalue adds a Lua string that is on the top of the stack; luaL_addlstring adds strings with an explicit length; luaL_addstring adds zero-terminated strings; and luaL_addchar adds single characters. These functions have the following prototypes:

```
void luaL_buffinit   (lua_State *L, luaL_Buffer *B);
void luaL_addvalue   (luaL_Buffer *B);
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
void luaL_addstring  (luaL_Buffer *B, const char *s);
```

[1] The directive p was introduced in Lua 5.2. The directives I and U were introduced in Lua 5.3.

[2] These two functions were introduced in Lua 5.2.

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

Figure 30.4, "A simplified implementation for `table.concat`" illustrates the use of these functions with a simplified implementation of the function `table.concat`.

Figure 30.4. A simplified implementation for `table.concat`

```
static int tconcat (lua_State *L) {
  luaL_Buffer b;
  int i, n;
  luaL_checktype(L, 1, LUA_TTABLE);
  n = luaL_len(L, 1);
  luaL_buffinit(L, &b);
  for (i = 1; i <= n; i++) {
    lua_geti(L, 1, i);  /* get string from table */
    luaL_addvalue(b);   /* add it to the buffer */
  }
  luaL_pushresult(&b);
  return 1;
}
```

In that function, we first call `luaL_buffinit` to initialize the buffer. We then add elements to the buffer one by one, in this example using `luaL_addvalue`. Finally, `luaL_pushresult` flushes the buffer and leaves the final string on the top of the stack.

When we use the auxlib buffer, we have to worry about one detail. After we initialize a buffer, it may keep some internal data in the Lua stack. Therefore, we cannot assume that the stack top will remain where it was before we started using the buffer. Moreover, although we can use the stack for other tasks while using a buffer, the push/pop count for these uses must be balanced every time we access the buffer. The only exception to this rule is `luaL_addvalue`, which assumes that the string to be added to the buffer is on the top of the stack.

Storing State in C Functions

Frequently, C functions need to keep some non-local data, that is, data that outlive their invocation. In C, we typically use global (`extern`) or static variables for this need. When we are programming library functions for Lua, however, neither works well. First, we cannot store a generic Lua value in a C variable. Second, a library that uses such variables will not work with multiple Lua states.

A better approach is to get some help from Lua. A Lua function has two places to store non-local data: global variables and non-local variables. The C API offers two similar places to store non-local data: the registry and upvalues.

The registry

The *registry* is a global table that can be accessed only by C code.[3] Typically, we use it to store data to be shared among several modules.

The registry is always located at the *pseudo-index* `LUA_REGISTRYINDEX`. A pseudo-index is like an index into the stack, except that its associated value is not on the stack. Most functions in the Lua API

[3] Actually, we can access it from Lua using the debug function `debug.getregistry`, but we really should not use this function except for debugging.

that accept indices as arguments also accept pseudo-indices —the exceptions being those functions that manipulate the stack itself, such as lua_remove and lua_insert. For instance, to get a value stored with key "Key" in the registry, we can use the following call:

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

The registry is a regular Lua table. As such, we can index it with any non-nil Lua value. However, because all C modules share the same registry, we must choose with care what values we use as keys, to avoid collisions. String keys are particularly useful when we want to allow other independent libraries to access our data, because all they need to know is the key name. For those keys, there is no bulletproof method of choosing names, but there are some good practices, such as avoiding common names and prefixing our names with the library name or something like it. (Prefixes like lua or lualib are not good choices.)

We should never use our own numbers as keys in the registry, because Lua reserves numeric keys for its *reference system*. This system comprises a pair of functions in the auxiliary library that allow us to store values in a table without worrying about how to create unique keys. The function luaL_ref creates new references:

```
int ref = luaL_ref(L, LUA_REGISTRYINDEX);
```

The previous call pops a value from the stack, stores it into the registry with a fresh integer key, and returns this key. We call this key a *reference*.

As the name implies, we use references mainly when we need to store a reference to a Lua value inside a C structure. As we have seen, we should never store pointers to Lua strings outside the C function that retrieved them. Moreover, Lua does not even offer pointers to other objects, such as tables or functions. So, we cannot refer to Lua objects through pointers. Instead, when we need such pointers, we create a reference and store it in C.

To push the value associated with a reference ref onto the stack, we simply write this:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, ref);
```

Finally, to release both the value and the reference, we call luaL_unref:

```
luaL_unref(L, LUA_REGISTRYINDEX, ref);
```

After this call, a new call to luaL_ref may return this reference again.

The reference system treats nil as a special case. Whenever we call luaL_ref for a nil value, it does not create a new reference, but instead returns the constant reference LUA_REFNIL. The following call has no effect:

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

The next one pushes a nil, as expected:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

The reference system also defines the constant LUA_NOREF, which is an integer different from any valid reference. It is useful to signal that a value treated as a reference is invalid.

When we create a Lua state, the registry comes with two predefined references:

LUA_RIDX_MAINTHREAD keeps the Lua state itself, which is also its main thread.

`LUA_RIDX_GLOBALS` keeps the global environment.

Another safe way to create unique keys in the registry is to use as key the address of a static variable in our code: The C link editor ensures that this key is unique across all loaded libraries. To use this option, we need the function `lua_pushlightuserdata`, which pushes on the stack a value representing a C pointer. The following code shows how to store and retrieve a string from the registry using this method:

```
/* variable with a unique address */
static char Key = 'k';

/* store a string */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_pushstring(L, myStr);  /* push value */
lua_settable(L, LUA_REGISTRYINDEX);  /* registry[&Key] = myStr */

/* retrieve a string */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_gettable(L, LUA_REGISTRYINDEX);  /* retrieve value */
myStr = lua_tostring(L, -1);  /* convert to string */
```

We will discuss light userdata in more detail in the section called “Light Userdata”.

To simplify the use of variable addresses as unique keys, Lua 5.2 introduced two new functions: `lua_rawgetp` and `lua_rawsetp`. They are similar to `lua_rawgeti` and `lua_rawseti`, but they use C pointers (translated to light userdata) as keys. With them, we can write the previous code like this:

```
static char Key = 'k';

/* store a string */
lua_pushstring(L, myStr);
lua_rawsetp(L, LUA_REGISTRYINDEX, (void *)&Key);

/* retrieve a string */
lua_rawgetp(L, LUA_REGISTRYINDEX, (void *)&Key);
myStr = lua_tostring(L, -1);
```

Both functions use raw accesses. As the registry does not have a metatable, a raw access has the same behavior as a regular access, and it is slightly more efficient.

Upvalues

While the registry offers global variables, the *upvalue* mechanism implements an equivalent of C static variables that are visible only inside a particular function. Every time we create a new C function in Lua, we can associate with it any number of upvalues, each one holding a single Lua value. Later, when we call the function, it has free access to any of its upvalues, using pseudo-indices.

We call this association of a C function with its upvalues a *closure*. A C closure is a C approximation to a Lua closure. In particular, we can create different closures using the same function code, but with different upvalues.

To see a simple example, let us create a function `newCounter` in C. (We defined a similar function in Lua in Chapter 9, *Closures*.) This function is a factory: it returns a new counter function each time it is called, as in this example:

```
c1 = newCounter()
```

```
print(c1(), c1(), c1())   --> 1    2    3
c2 = newCounter()
print(c2(), c2(), c1())   --> 1    2    4
```

Although all counters share the same C code, each one keeps its own independent counter. The factory function is like this:

```
static int counter (lua_State *L);  /* forward declaration */

int newCounter (lua_State *L) {
  lua_pushinteger(L, 0);
  lua_pushcclosure(L, &counter, 1);
  return 1;
}
```

The key function here is `lua_pushcclosure`, which creates a new closure. Its second argument is the base function (`counter`, in the example) and the third is the number of upvalues (1, in the example). Before creating a new closure, we must push on the stack the initial values for its upvalues. In our example, we push zero as the initial value for the single upvalue. As expected, `lua_pushcclosure` leaves the new closure on the stack, so the closure is ready to be returned as the result of `newCounter`.

Now, let us see the definition of `counter`:

```
static int counter (lua_State *L) {
  int val = lua_tointeger(L, lua_upvalueindex(1));
  lua_pushinteger(L, ++val);  /* new value */
  lua_copy(L, -1, lua_upvalueindex(1));  /* update upvalue */
  return 1;  /* return new value */
}
```

Here, the key element is the macro `lua_upvalueindex`, which produces the pseudo-index of an upvalue. In particular, the expression `lua_upvalueindex(1)` gives the pseudo-index of the first upvalue of the running function. Again, this pseudo-index is like any stack index, except that it does not live on the stack. So, the call to `lua_tointeger` retrieves the current value of the first (and only) upvalue as an integer. Then, the function `counter` pushes the new value `++val`, copies it as the new upvalue's value, and returns it.

As a more advanced example, we will implement tuples using upvalues. A *tuple* is a kind of constant structure with anonymous fields; we can retrieve a specific field with a numerical index, or we can retrieve all fields at once. In our implementation, we represent tuples as functions that store their values in their upvalues. When called with a numerical argument, the function returns that specific field. When called without arguments, it returns all its fields. The following code illustrates the use of tuples:

```
x = tuple.new(10, "hi", {}, 3)
print(x(1))     --> 10
print(x(2))     --> hi
print(x())      --> 10  hi  table: 0x8087878  3
```

In C, we will represent all tuples by the same function `t_tuple`, presented in Figure 30.5, “An implementation of tuples”.

Figure 30.5. An implementation of tuples

```
#include "lauxlib.h"

int t_tuple (lua_State *L) {
  lua_Integer op = luaL_optinteger(L, 1, 0);
  if (op == 0) {  /* no arguments? */
    int i;
    /* push each valid upvalue onto the stack */
    for (i = 1; !lua_isnone(L, lua_upvalueindex(i)); i++)
       lua_pushvalue(L, lua_upvalueindex(i));
    return i - 1;  /* number of values */
  }
  else {  /* get field 'op' */
    luaL_argcheck(L, 0 < op && op <= 256, 1,
                     "index out of range");
    if (lua_isnone(L, lua_upvalueindex(op)))
      return 0;  /* no such field */
    lua_pushvalue(L, lua_upvalueindex(op));
    return 1;
  }
}

int t_new (lua_State *L) {
  int top = lua_gettop(L);
  luaL_argcheck(L, top < 256, top, "too many fields");
  lua_pushcclosure(L, t_tuple, top);
  return 1;
}

static const struct luaL_Reg tuplelib [] = {
  {"new", t_new},
  {NULL, NULL}
};

int luaopen_tuple (lua_State *L) {
  luaL_newlib(L, tuplelib);
  return 1;
}
```

Because we can call a tuple with or without a numeric argument, `t_tuple` uses `luaL_optinteger` to get its optional argument. This function is similar to `luaL_checkinteger`, but it does not complain if the argument is absent; instead, it returns a given default value (0, in the example).

The maximum number of upvalues to a C function is 255, and the maximum index we can use with `lua_upvalueindex` is 256. So, we use `luaL_argcheck` to ensure these limits.

When we index a non-existent upvalue, the result is a pseudo-value whose type is `LUA_TNONE`. (When we access a stack index above the current top, we also get a pseudo-value with this type `LUA_TNONE`.) Our function `t_tuple` uses `lua_isnone` to test whether it has a given upvalue. However, we should never use `lua_upvalueindex` with a negative index or with an index greater than 256 (which is one plus the maximum number of upvalues for a C function), so we must check for this condition when the user provides the index. The function `luaL_argcheck` checks a given condition, raising an error with a nice message if the condition fails:

```
> t = tuple.new(2, 4, 5)
> t(300)
  --> stdin:1: bad argument #1 to 't' (index out of range)
```

The third argument to `luaL_argcheck` provides the argument number for the error message (1, in the example), and the fourth argument provides a complement to the message (`"index out of range"`).

The function to create tuples, `t_new` (also in Figure 30.5, "An implementation of tuples"), is trivial: because its arguments are already on the stack, it first checks that the number of fields respects the limit for upvalues in a closure and then call `lua_pushcclosure` to create a closure of `t_tuple` with all its arguments as upvalues. Finally, the array `tuplelib` and the function `luaopen_tuple` (also in Figure 30.5, "An implementation of tuples") are the standard code to create a library `tuple` with that single function `new`.

Shared upvalues

Often, we need to share some values or variables among all functions in a library. Although we can use the registry for that task, we can also use upvalues.

Unlike Lua closures, C closures cannot share upvalues. Each closure has its own independent upvalues. However, we can set the upvalues of different functions to refer to a common table, so that this table becomes a common environment where the functions can share data.

Lua offers a function that eases the task of sharing an upvalue among all functions of a library. We have been opening C libraries with `luaL_newlib`. Lua implements this function as the following macro:

```
#define luaL_newlib(L,lib)  \
    (luaL_newlibtable(L,lib), luaL_setfuncs(L,lib,0))
```

The macro `luaL_newlibtable` just creates a new table for the library. (This table has a preallocated size equal to the number of functions in the given library.) The function `luaL_setfuncs` then adds the functions in the list `lib` to that new table, which is on the top of the stack.

The third parameter to `luaL_setfuncs` is what we are interested in here. It gives the number of shared upvalues the new functions in the library will have. The initial values for these upvalues should be on the stack, as happens with `lua_pushcclosure`. Therefore, to create a library where all functions share a common table as their single upvalue, we can use the following code:

```
/* create library table ('lib' is its list of functions) */
luaL_newlibtable(L, lib);
/* create shared upvalue */
lua_newtable(L);
/* add functions in list 'lib' to the new library, sharing
   previous table as upvalue */
luaL_setfuncs(L, lib, 1);
```

The last call also removes the shared table from the stack, leaving there only the new library.

Exercises

Exercise 30.1: Implement a filter function in C. It should receive a list and a predicate and return a new list with all elements from the given list that satisfy the predicate:

```
t = filter({1, 3, 20, -4, 5}, function (x) return x < 5 end)
-- t = {1, 3, -4}
```

(A predicate is just a function that tests some condition, returning a Boolean.)

Exercise 30.2: Modify the function `l_split` (from Figure 30.2, "Splitting a string") so that it can work with strings containing zeros. (Among other changes, it should use `memchr` instead of `strchr`.)

Exercise 30.3: Reimplement the function `transliterate` (Exercise 10.3) in C.

Exercise 30.4: Implement a library with a modification of `transliterate` so that the transliteration table is not given as an argument, but instead is kept by the library. Your library should offer the following functions:

```
lib.settrans (table)   -- set the transliteration table
lib.gettrans ()        -- get the transliteration table
lib.transliterate(s)    -- transliterate 's' according to the
                          current table
```

Use the registry to keep the transliteration table.

Exercise 30.5: Repeat the previous exercise using an upvalue to keep the transliteration table.

Exercise 30.6: Do you think it is a good design to keep the transliteration table as part of the state of the library, instead of being a parameter to `transliterate`?

EXTEND
- FUNCTIONS
- TYPES

Chapter 31. User-Defined Types in C

In the previous chapter, we saw how to extend Lua with new functions written in C. Now, we will see how to extend Lua with new types written in C. We will start with a small example; through the chapter, we will extend it with metamethods and other goodies.

Our running example in this chapter will be a quite simple type: Boolean arrays. The main motivation for this example is that it does not involve complex algorithms, so we can concentrate on API issues. Nevertheless, the example is useful by itself. Of course, we can use tables to implement arrays of Booleans in Lua. But a C implementation, where we store each entry in one single bit, uses less than 3% of the memory used by a table.

Our implementation will need the following definitions:

```
#include <limits.h>

#define BITS_PER_WORD (CHAR_BIT * sizeof(unsigned int))
#define I_WORD(i)     ((unsigned int)(i) / BITS_PER_WORD)
#define I_BIT(i)      (1 << ((unsigned int)(i) % BITS_PER_WORD))
```

`BITS_PER_WORD` is the number of bits in an unsigned integer. The macro `I_WORD` computes the word that stores the bit corresponding to a given index, and `I_BIT` computes a mask to access the correct bit inside this word.

We will represent our arrays with the following structure:

```
typedef struct BitArray {
  int size;
  unsigned int values[1];  /* variable part */
} BitArray;
```

We declare the array `values` with size 1 only as a placeholder, because C 89 does not allow an array with size 0; we will set the actual size when we allocate the array. The next expression computes the total size for an array with `n` elements:

```
sizeof(BitArray) + I_WORD(n - 1) * sizeof(unsigned int)
```

(We subtract one from `n` because the original structure already includes space for one element.)

Userdata

In this first version, we will use explicit calls to set and get values, as in the next example:

```
a = array.new(1000)
for i = 1, 1000 do
  array.set(a, i, i % 2 == 0)     -- a[i] = (i % 2 == 0)
end
print(array.get(a, 10))           --> true
print(array.get(a, 11))           --> false
print(array.size(a))              --> 1000
```

self!

array.get(a,10)
vs
a:get(10)

Later we will see how to support both an object-oriented style, like `a:get(i)`, and a conventional syntax, like `a[i]`. For all versions, the underlying functions are the same, defined in Figure 31.1, “Manipulating a Boolean array”.

Figure 31.1. Manipulating a Boolean array

```
static int newarray (lua_State *L) {
  int i;
  size_t nbytes;
  BitArray *a;

  int n = (int)luaL_checkinteger(L, 1);   /* number of bits */
  luaL_argcheck(L, n >= 1, 1, "invalid size");
  nbytes = sizeof(BitArray) + I_WORD(n - 1)*sizeof(unsigned int);
  a = (BitArray *)lua_newuserdata(L, nbytes);

  a->size = n;
  for (i = 0; i <= I_WORD(n - 1); i++)
    a->values[i] = 0;  /* initialize array */

  return 1;  /* new userdata is already on the stack */
}

static int setarray (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  int index = (int)luaL_checkinteger(L, 2) - 1;

  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                   "index out of range");
  luaL_checkany(L, 3);

  if (lua_toboolean(L, 3))
    a->values[I_WORD(index)] |= I_BIT(index);  /* set bit */
  else
    a->values[I_WORD(index)] &= ~I_BIT(index);  /* reset bit */
  return 0;
}

static int getarray (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  int index = (int)luaL_checkinteger(L, 2) - 1;

  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                   "index out of range");

  lua_pushboolean(L, a->values[I_WORD(index)] & I_BIT(index));
  return 1;
}
```

Let us see them, bit by bit.

Our first concern is how to represent a C structure in Lua. Lua provides a basic type specifically for this task, called *userdata*. A userdata offers a raw memory area, with no predefined operations in Lua, which we can use to store anything.

The function `lua_newuserdata` allocates a block of memory with a given size, pushes the corresponding userdata on the stack, and returns the block address:

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

If for some reason we need to allocate memory by other means, it is very easy to create a userdata with the size of a pointer and to store there a pointer to the real memory block. We will see examples of this technique in Chapter 32, *Managing Resources*.

Our first function in Figure 31.1, “Manipulating a Boolean array”, `newarray`, uses `lua_newuserdata` to create new arrays. Its code is straightforward. It checks its sole parameter (the array size, in bits), computes the array size in bytes, creates a userdata with the appropriate size, initializes its fields, and returns the userdata to Lua.

The next function is `setarray`, which receives three arguments: the array, the index, and the new value. It assumes that indices start at one, as usual in Lua. Because Lua accepts any value for a Boolean, we use `luaL_checkany` for the third parameter: it ensures only that there is a value (any value) for this parameter. If we call `setarray` with bad arguments, we get explanatory error messages, as in the following examples:

```
array.set(0, 11, 0)
  --> stdin:1: bad argument #1 to 'set' ('array' expected)
array.set(a, 1)
  --> stdin:1: bad argument #3 to 'set' (value expected)
```

The last function in Figure 31.1, “Manipulating a Boolean array” is `getarray`, the function to retrieve an entry. It is similar to `setarray`.

We will also define a function to retrieve the size of an array and some extra code to initialize our library; see Figure 31.2, “Extra code for the Boolean array library”.

Figure 31.2. Extra code for the Boolean array library

```
static int getsize (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  lua_pushinteger(L, a->size);
  return 1;
}

static const struct luaL_Reg arraylib [] = {
  {"new", newarray},
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_newlib(L, arraylib);
  return 1;
}
```

INIT LIB

Again, we use `luaL_newlib`, from the auxiliary library. It creates a table and fills it with the pairs name–function specified by the array `arraylib`.

Metatables

userdata

Our current implementation has a major vulnerability. Suppose the user writes something like `array.set(io.stdin, 1, false)`. The value of `io.stdin` is a userdata with a pointer to a stream (`FILE *`). Because it is a userdata, `array.set` will gladly accept it as a valid argument; the probable result will be a memory corruption (with luck we will get an index-out-of-range error instead). Such behavior is unacceptable for any Lua library. No matter how we use a library, it should neither corrupt C data nor cause the Lua system to crash.

The usual method to distinguish one type of userdata from another is to create a unique metatable for that type. Every time we create a userdata, we mark it with the corresponding metatable; every time we get a userdata, we check whether it has the right metatable. Because Lua code cannot change the metatable of a userdata, it cannot deceive these checks.

We also need a place to store this new metatable, so that we can access it to create new userdata and to check whether a given userdata has the correct type. As we saw earlier, there are two options for storing the metatable: in the registry or as an upvalue for the functions in the library. It is customary, in Lua, to register any new C type into the registry, using a *type name* as the index and the metatable as the value. As with any other registry index, we must choose a type name with care, to avoid clashes. Our example will use the name `"LuaBook.array"` for its new type.

registry

"namespace?"

As usual, the auxiliary library offers some functions to help us here. The new auxiliary functions we will use are these:

```
int   luaL_newmetatable (lua_State *L, const char *tname);
void  luaL_getmetatable (lua_State *L, const char *tname);
void *luaL_checkudata   (lua_State *L, int index,
                                       const char *tname);
```

✓

The function `luaL_newmetatable` creates a new table (to be used as a metatable), leaves the new table on the top of the stack, and maps the table to the given name in the registry. The function `luaL_getmetatable` retrieves the metatable associated with `tname` from the registry. Finally, `luaL_checkudata` checks whether the object at the given stack position is a userdata with a metatable that matches the given name. It raises an error if the object is not a userdata or if it does not have the correct metatable; otherwise, it returns the userdata address.

Now we can start our modifications. The first step is to change the function that opens the library so that it creates the metatable for arrays:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_newlib(L, arraylib);
  return 1;
}
```

CREATES METATABLES WHEN LIB IS OPENED

The next step is to change `newarray` so that it sets this metatable in all arrays that it creates:

```
static int newarray (lua_State *L) {

  as before

  luaL_getmetatable(L, "LuaBook.array");
  lua_setmetatable(L, -2);
```

ADD @ END

```
  return 1;  /* new userdata is already on the stack */
}
```

The function `lua_setmetatable` pops a table from the stack and sets it as the metatable of the object at the given index. In our case, this object is the new userdata.

Finally, `setarray`, `getarray`, and `getsize` have to check whether they have got a valid array as their first argument. To simplify their tasks, we define the following macro:

```
#define checkarray(L) \
          (BitArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

Using this macro, the new definition for `getsize` is straightforward:

```
static int getsize (lua_State *L) {
  BitArray *a = checkarray(L);
  lua_pushinteger(L, a->size);
  return 1;
}
```

Because `setarray` and `getarray` also share code to read and check the index as their second argument, we factor out their common parts in a new auxiliary function (`getparams`).

Figure 31.3. New versions for `setarray`/`getarray`

```
static unsigned int *getparams (lua_State *L,
                                unsigned int *mask) {
  BitArray *a = checkarray(L);
  int index = (int)luaL_checkinteger(L, 2) - 1;

  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                   "index out of range");

  *mask = I_BIT(index);  /* mask to access correct bit */
  return &a->values[I_WORD(index)]; /* word address */
}

static int setarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getparams(L, &mask);
  luaL_checkany(L, 3);
  if (lua_toboolean(L, 3))
    *entry |= mask;
  else
    *entry &= ~mask;

  return 0;
}

static int getarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getparams(L, &mask);
  lua_pushboolean(L, *entry & mask);
  return 1;
}
```

With this new function, `setarray` and `getarray` are straightforward, see Figure 31.3, “New versions for `setarray`/`getarray`”. Now, if we call them with an invalid userdata, we will get a proper error message:

```
a = array.get(io.stdin, 10)
--> bad argument #1 to 'get' (LuaBook.array expected, got FILE*)
```

Object-Oriented Access

Our next step is to transform our new type into an object, so that we can operate on its instances using the usual object-oriented syntax, like this:

```
a = array.new(1000)
print(a:size())     --> 1000
a:set(10, true)
print(a:get(10))    --> true
```

Remember that `a:size()` is equivalent to `a.size(a)`. Therefore, we have to arrange for the expression `a.size` to return our function `getsize`. The key mechanism here is the `__index` metamethod. For tables, Lua calls this metamethod whenever it cannot find a value for a given key. For userdata, Lua calls it in every access, because userdata have no keys at all.

Assume that we run the following code:

```
do
  local metaarray = getmetatable(array.new(1))
  metaarray.__index = metaarray
  metaarray.set = array.set
  metaarray.get = array.get
  metaarray.size = array.size
end
```

mapping

In the first line, we create an array only to get its metatable, which we assign to `metaarray`. (We cannot set the metatable of a userdata from Lua, but we can get it.) Then we set `metaarray.__index` to `metaarray`. When we evaluate `a.size`, Lua cannot find the key `"size"` in the object `a`, because the object is a userdata. Therefore, Lua tries to get this value from the field `__index` of the metatable of `a`, which happens to be `metaarray` itself. But `metaarray.size` is `array.size`, so `a.size(a)` results in `array.size(a)`, as we wanted.

Of course, we can write the same thing in C. We can do even better: now that arrays are objects, with their own operations, we do not need to have these operations in the table `array` anymore. The only function that our library still has to export is `new`, to create new arrays. All other operations come only as methods. The C code can register them directly as such.

The operations `getsize`, `getarray`, and `setarray` do not change from our previous approach. What will change is how we register them. That is, we have to change the code that opens the library. First, we need two separate function lists: one for regular functions and one for methods.

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};
```

The new version of the open function `luaopen_array` has to create the metatable, assign it to its own `__index` field, register all the methods there, and create and fill the `array` table:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");  /* create metatable */
  lua_pushvalue(L, -1);  /* duplicate the metatable */
  lua_setfield(L, -2, "__index");  /* mt.__index = mt */
  luaL_setfuncs(L, arraylib_m, 0);  /* register metamethods */
  luaL_newlib(L, arraylib_f);  /* create lib table */
  return 1;
}
```

Here we use `luaL_setfuncs` again, to set the functions from the list `arraylib_m` into the metatable, which is on the top of the stack. Then we call `luaL_newlib` to create a new table and register the functions from the list `arraylib_f` there.

As a final touch, we will add a `__tostring` method to our new type, so that `print(a)` prints `"array"` plus the size of the array inside parentheses. The function itself is here:

```
int array2string (lua_State *L) {
  BitArray *a = checkarray(L);
  lua_pushfstring(L, "array(%d)", a->size);
  return 1;
}
```

The call to `lua_pushfstring` formats the string and leaves it on the top of the stack. We also have to add `array2string` to the list `arraylib_m`, to include it in the metatable of array objects:

```
static const struct luaL_Reg arraylib_m [] = {
  {"__tostring", array2string},
  other methods
};
```

Array Access

A better alternative to the object-oriented notation is to use a regular array notation to access our arrays. Instead of writing `a:get(i)`, we could simply write `a[i]`. For our example, this is easy to do, because our functions `setarray` and `getarray` already receive their arguments in the order that they are given to the corresponding metamethods. A quick solution is to define these metamethods directly in Lua:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size
```

(We must run this code on the original implementation for arrays, without the modifications for object-oriented access.) That is all we need to use the standard syntax:

```
a = array.new(1000)
a[10] = true         -- 'setarray'
```

```
print(a[10])         -- 'getarray'   --> true
print(#a)            -- 'getsize'    --> 1000
```

If we prefer, we can register these metamethods in our C code. For this, we again modify our initialization function; see Figure 31.4, “New initialization code for the Bit Array library”.

Figure 31.4. New initialization code for the Bit Array library

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
  {"__newindex", setarray},
  {"__index", getarray},
  {"__len", getsize},
  {"__tostring", array2string},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_setfuncs(L, arraylib_m, 0);
  luaL_newlib(L, arraylib_f);
  return 1;
}
```

In this new version, again we have only one public function, `new`. All other functions are available only as metamethods for specific operations.

Light Userdata

The kind of userdata that we have been using until now is called *full userdata*. Lua offers another kind of userdata, called *light userdata*.

A light userdata is a value that represents a C pointer, that is, a `void *` value. A light userdata is a value, not an object; we do not create them (in the same way that we do not create numbers). To put a light userdata onto the stack, we call `lua_pushlightuserdata`:

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

Despite their common name, light userdata and full userdata are quite different things. Light userdata are not buffers, but bare pointers. They have no metatables. Like numbers, light userdata are not managed by the garbage collector.

Sometimes, people use light userdata as a cheap alternative to full userdata. This is not a typical use, however. First, light userdata do not have metatables, so there is no way to know their types. Second, despite the name, full userdata are inexpensive, too. They add little overhead compared to a `malloc` for the given memory size.

The real use of light userdata comes from equality. As a full userdata is an object, it is only equal to itself. A light userdata, on the other hand, represents a C pointer value. As such, it is equal to any userdata that represents the same pointer. Therefore, we can use light userdata to find C objects inside Lua.

We have already seen a typical use of light userdata, as keys in the registry (the section called “The registry”). There, the equality of light userdata was fundamental. Every time we push the same address with `lua_pushlightuserdata`, we get the same Lua value and, therefore, the same entry in the registry.

Another typical scenario in Lua is to have Lua objects acting as proxies to corresponding C objects. For instance, the I/O library uses Lua userdata to represent C streams inside Lua. When the action goes from Lua to C, the mapping from the Lua object to the C object is easy. Again using the example of the I/O library, each Lua stream keeps a pointer to its corresponding C stream. However, when the action goes from C to Lua, the mapping can be tricky. As an example, suppose we have some kind of callback in our I/O system (e.g., to tell that there is data to be read). The callback receives the C stream where is should operate. From that, how can we find its corresponding Lua object? Because the C stream is defined by the C standard library, not by us, we cannot store anything there.

Light userdata provide a nice solution for this mapping. We keep a table where the indices are light userdata with the stream addresses, and the values are the full userdata that represent the streams in Lua. In a callback, once we have a stream address, we use it —as a light userdata— as an index into that table to retrieve its corresponding Lua object. (The table should probably have weak values; otherwise, those full userdata would never be collected.)

Exercises

Exercise 31.1: Modify the implementation of `setarray` so that it accepts only Boolean values.

Exercise 31.2: We can see a Boolean array as a set of integers (the indices with true values in the array). Add to the implementation of Boolean arrays functions to compute the union and intersection of two arrays. These functions should receive two arrays and return a new one, without modifying its parameters.

Exercise 31.3: Extend the previous exercise so that we can use addition to get the union of two arrays and multiplication for the intersection.

Exercise 31.4: Modify the implementation of the `__tostring` metamethod so that it shows the full contents of the array in an appropriate way. Use the buffer facility (the section called “String Manipulation”) to create the resulting string.

Exercise 31.5: Based on the example for Boolean arrays, implement a small C library for integer arrays.

Chapter 32. Managing Resources

In our implementation of Boolean arrays in the previous chapter, we did not need to worry about managing resources. Those arrays need only memory. Each userdata representing an array has its own memory, which is managed by Lua. When an array becomes garbage (that is, inaccessible by the program), Lua eventually collects it and frees its memory.

Life is not always that easy. Sometimes, an object needs other resources besides raw memory, such as file descriptors, window handles, and the like. (Often these resources are just memory too, but managed by some other part of the system.) In such cases, when the object becomes garbage and is collected, somehow these other resources must be released too.

As we saw in the section called "Finalizers", Lua provides finalizers in the form of the `__gc` metamethod. To illustrate the use of this metamethod in C and of the API as a whole, in this chapter we will develop two Lua bindings for external facilities. The first example is another implementation for a function to traverse a directory. The second (and more substantial) example is a binding to *Expat*, an open source XML parser.

A Directory Iterator

In the section called "C Functions", we implemented a function `dir` to traverse directories that returned a table with all files from a given directory. Our new implementation will return an iterator that returns a new entry each time it is called. With this new implementation, we will be able to traverse a directory with a loop like this:

```
for fname in dir.open(".") do
  print(fname)
end
```

To iterate over a directory, in C, we need a `DIR` structure. Instances of `DIR` are created by `opendir` and must be explicitly released with a call to `closedir`. Our previous implementation kept its `DIR` instance as a local variable and closed this instance after retrieving the last file name. Our new implementation cannot keep this `DIR` instance in a local variable, because it must query this value over several calls. Moreover, it cannot close the directory only after retrieving the last name; if the program breaks the loop, the iterator will never retrieve this last name. Therefore, to make sure that the `DIR` instance is always released, we will store its address in a userdata and use the `__gc` metamethod of this userdata to release the directory structure.

Despite its central role in our implementation, this userdata representing a directory does not need to be visible to Lua. The function `dir.open` returns an iterator function, and this function is what Lua sees. The directory can be an upvalue of the iterator function. As such, the iterator function has direct access to this structure, but Lua code does not (and does not need to).

In all, we need three C functions. First, we need the function `dir.open`, a factory function that Lua calls to create iterators; it must open a `DIR` structure and create a closure of the iterator function with this structure as an upvalue. Second, we need the iterator function. Third, we need the `__gc` metamethod, which closes a `DIR` structure. As usual, we also need an extra function to make initial arrangements, such as to create and initialize a metatable for directories.

Let us start our code with the function `dir.open`, shown in Figure 32.1, "The `dir.open` factory function".

Figure 32.1. The `dir.open` factory function

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

#include "lua.h"
#include "lauxlib.h"

/* forward declaration for the iterator function */
static int dir_iter (lua_State *L);

static int l_dir (lua_State *L) {
  const char *path = luaL_checkstring(L, 1);

  /* create a userdata to store a DIR address */
  DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

  /* pre-initialize it */
  *d = NULL;

  /* set its metatable */
  luaL_getmetatable(L, "LuaBook.dir");
  lua_setmetatable(L, -2);

  /* try to open the given directory */
  *d = opendir(path);
  if (*d == NULL)  /* error opening the directory? */
    luaL_error(L, "cannot open %s: %s", path, strerror(errno));

  /* creates and returns the iterator function;
     its sole upvalue, the directory userdata,
     is already on the top of the stack */
  lua_pushcclosure(L, dir_iter, 1);
  return 1;
}
```

A subtle point in this function is that it must create the userdata before opening the directory. If it first opens the directory, and then the call to `lua_newuserdata` raises a memory error, the function loses and leaks the `DIR` structure. With the correct order, the `DIR` structure, once created, is immediately associated with the userdata; whatever happens after that, the `__gc` metamethod will eventually release the structure.

Another subtle point is the consistency of the userdata. Once we set its metatable, the `__gc` metamethod will be called no matter what. So, before setting the metatable, we pre-initialize the userdata with `NULL` to ensure that it has some well-defined value.

The next function is `dir_iter` (in Figure 32.2, “Other functions for the `dir` library”), the iterator itself.

Figure 32.2. Other functions for the `dir` library

```
static int dir_iter (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, lua_upvalueindex(1));
  struct dirent *entry = readdir(d);
  if (entry != NULL) {
    lua_pushstring(L, entry->d_name);
    return 1;
  }
  else return 0;  /* no more values to return */
}

static int dir_gc (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, 1);
  if (d) closedir(d);
  return 0;
}

static const struct luaL_Reg dirlib [] = {
  {"open", l_dir},
  {NULL, NULL}
};

int luaopen_dir (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.dir");

  /* set its __gc field */
  lua_pushcfunction(L, dir_gc);
  lua_setfield(L, -2, "__gc");

  /* create the library */
  luaL_newlib(L, dirlib);
  return 1;
}
```

Its code is straightforward. It gets the `DIR` structure's address from its upvalue and calls `readdir` to read the next entry.

The function `dir_gc` (also in Figure 32.2, “Other functions for the `dir` library”) is the `__gc` metamethod. This metamethod closes a directory. As we mentioned before, it must take one precaution: in case of errors in the initialization, the directory can be `NULL`.

The last function in Figure 32.2, “Other functions for the `dir` library”, `luaopen_dir`, is the function that opens this one-function library.

This complete example has an interesting subtlety. At first, it may seem that `dir_gc` should check whether its argument is a directory and whether it has not been closed already. Otherwise, a malicious user could call it with another kind of userdata (a file, for instance) or finalize a directory twice, with disastrous consequences. However, there is no way for a Lua program to access this function: it is stored only in the metatable of directories, which in turn are stored as upvalues of the iteration functions. Lua programs cannot access these directories.

An XML Parser

Now we will look at a simplified implementation of a Lua binding for Expat, which we will call `lxp`. Expat is an open source XML 1.0 parser written in C. It implements SAX, the *Simple API for XML*. SAX is an event-based API. This means that a SAX parser reads an XML document and, as it goes, reports to the application what it finds, through callbacks. For instance, if we instruct Expat to parse a string like `"<tag cap="5">hi</tag>"`, it will generate three events: a *start-element* event, when it reads the substring `"<tag cap="5">"`; a *text* event (also called a *character data* event), when it reads `"hi"`; and an *end-element* event, when it reads `"</tag>"`. Each of these events calls an appropriate *callback handler* in the application.

Here we will not cover the entire Expat library. We will concentrate only on those parts that illustrate new techniques for interacting with Lua. Although Expat handles more than a dozen different events, we will consider only the three events that we saw in the previous example (start elements, end elements, and text).[1]

The part of the Expat API that we need for this example is small. First, we need the functions to create and destroy an Expat parser:

```
XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

The `encoding` argument is optional; we will use `NULL` in our binding.

After we have a parser, we must register its callback handlers:

```
void XML_SetElementHandler(XML_Parser p,
                           XML_StartElementHandler start,
                           XML_EndElementHandler end);

void XML_SetCharacterDataHandler(XML_Parser p,
                                 XML_CharacterDataHandler hndl);
```

The first function registers handlers for start and end elements. The second function registers handlers for text (*character data*, in XML parlance).

All callback handlers take a user data as their first parameter. The start-element handler receives also the tag name and its attributes:

```
typedef void (*XML_StartElementHandler)(void *uData,
                                        const char *name,
                                        const char **atts);
```

The attributes come as a NULL-terminated array of strings, where each pair of consecutive strings holds an attribute name and its value. The end-element handler has only one extra parameter, the tag name:

```
typedef void (*XML_EndElementHandler)(void *uData,
                                      const char *name);
```

Finally, a text handler receives only the text as an extra parameter. This text string is not null-terminated; instead, it has an explicit length:

```
typedef void (*XML_CharacterDataHandler)(void *uData,
```

[1] The package `LuaExpat` offers a quite complete interface to Expat.

```
                                                   const char *s,
                                                   int len);
```

To feed text to Expat, we use the following function:

```
int XML_Parse (XML_Parser p, const char *s, int len, int isLast);
```

Expat receives the document to be parsed in pieces, through successive calls to the function `XML_Parse`. The last argument to `XML_Parse`, the Boolean `isLast`, informs Expat whether that piece is the last one of a document. This function returns zero if it detects a parse error. (Expat also provides functions to retrieve error information, but we will ignore them here, for the sake of simplicity.)

The last function we need from Expat allows us to set the user data that will be passed to the handlers:

```
void XML_SetUserData (XML_Parser p, void *uData);
```

Now let us have a look at how we can use this library in Lua. A first approach is a direct approach: simply export all those functions to Lua. A better approach is to adapt the functionality to Lua. For instance, because Lua is untyped, we do not need different functions to set each kind of callback. Better yet, we can avoid the callback registering functions altogether. Instead, when we create a parser, we give a callback table that contains all callback handlers, each with an appropriate key related to its corresponding event. For instance, if we want to print a layout of a document, we could use the following callback table:

```
local count = 0

callbacks = {
  StartElement = function (parser, tagname)
    io.write("+ ", string.rep("  ", count), tagname, "\n")
    count = count + 1
  end,

  EndElement = function (parser, tagname)
    count = count - 1
    io.write("- ", string.rep("  ", count), tagname, "\n")
  end,
}
```

Fed with the input `"<to> <yes/> </to>"`, these handlers would print this output:

```
+ to
+   yes
-   yes
- to
```

With this API, we do not need functions to manipulate callbacks. We manipulate them directly in the callback table. Thus, the whole API needs only three functions: one to create parsers, one to parse a piece of text, and one to close a parser. Actually, we will implement the last two functions as methods of parser objects. A typical use of the API could be like this:

```
local lxp = require "lxp"

p = lxp.new(callbacks)       -- create new parser

for l in io.lines() do       -- iterate over input lines
```

```
  assert(p:parse(l))          -- parse the line
  assert(p:parse("\n"))       -- add newline
end

assert(p:parse())             -- finish document
p:close()                     -- close parser
```

Now let us turn our attention to the implementation. The first decision is how to represent a parser in Lua. It is quite natural to use a userdata containing a C structure, but what do we need to put in it? We need at least the actual Expat parser and the callback table. We must also store a Lua state, because these parser objects are all that an Expat callback receives, and the callbacks need to call Lua. We can store the Expat parser and the Lua state (which are C values) directly in a C structure. For the callback table, which is a Lua value, one option is to create a reference to it in the registry and store that reference. (We will explore this option in Exercise 32.2). Another option is to use a *user value*. Each userdata can have one single Lua value directly associated with it; this value is called a user value.[2] With this option, the definition for a parser object is as follows:

```
#include <stdlib.h>
#include "expat.h"
#include "lua.h"
#include "lauxlib.h"

typedef struct lxp_userdata {
  XML_Parser parser;          /* associated expat parser */
  lua_State *L;
} lxp_userdata;
```

The next step is the function that creates parser objects, `lxp_make_parser`. Figure 32.3, “Function to create XML parser objects” shows its code.

[2] In Lua 5.2, this user value must be table.

Figure 32.3. Function to create XML parser objects

```
/* forward declarations for callback functions */
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts);
static void f_CharData (void *ud, const char *s, int len);
static void f_EndElement (void *ud, const char *name);

static int lxp_make_parser (lua_State *L) {
  XML_Parser p;

  /* (1) create a parser object */
  lxp_userdata *xpu = (lxp_userdata *)lua_newuserdata(L,
                                      sizeof(lxp_userdata));

  /* pre-initialize it, in case of error */
  xpu->parser = NULL;

  /* set its metatable */
  luaL_getmetatable(L, "Expat");
  lua_setmetatable(L, -2);

  /* (2) create the Expat parser */
  p = xpu->parser = XML_ParserCreate(NULL);
  if (!p)
    luaL_error(L, "XML_ParserCreate failed");

  /* (3) check and store the callback table */
  luaL_checktype(L, 1, LUA_TTABLE);
  lua_pushvalue(L, 1);  /* push table */
  lua_setuservalue(L, -2);   /* set it as the user value */

  /* (4) configure Expat parser */
  XML_SetUserData(p, xpu);
  XML_SetElementHandler(p, f_StartElement, f_EndElement);
  XML_SetCharacterDataHandler(p, f_CharData);
  return 1;
}
```

This function has four main steps:

- Its first step follows a common pattern: it first creates a userdata; then it pre-initializes the userdata with consistent values; and finally it sets its metatable. (The pre-initialization ensures that if there is any error during the initialization, the finalizer will find the userdata in a consistent state.)

- In step 2, the function creates an Expat parser, stores it in the userdata, and checks for errors.

- Step 3 ensures that the first argument to the function is actually a table (the callback table), and sets it as the user value for the new userdata.

- The last step initializes the Expat parser. It sets the userdata as the object to be passed to the callback functions and it sets the callback functions. Notice that these callback functions are the same for all parsers; after all, it is impossible to dynamically create new functions in C. Instead, those fixed C functions will use the callback table to decide which Lua functions they should call each time.

The next step is the parse method lxp_parse (Figure 32.4, “Function to parse an XML fragment”), which parses a piece of XML data.

Figure 32.4. Function to parse an XML fragment

```
static int lxp_parse (lua_State *L) {
  int status;
  size_t len;
  const char *s;
  lxp_userdata *xpu;

  /* get and check first argument (should be a parser) */
  xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* check if it is not closed */
  luaL_argcheck(L, xpu->parser != NULL, 1, "parser is closed");

  /* get second argument (a string) */
  s = luaL_optlstring(L, 2, NULL, &len);

  /* put callback table at stack index 3 */
  lua_settop(L, 2);
  lua_getuservalue(L, 1);

  xpu->L = L;  /* set Lua state */

  /* call Expat to parse string */
  status = XML_Parse(xpu->parser, s, (int)len, s == NULL);

  /* return error code */
  lua_pushboolean(L, status);
  return 1;
}
```

It gets two arguments: the parser object (the *self* of the method) and an optional piece of XML data. When called without any data, it informs Expat that the document has no more parts.

When lxp_parse calls XML_Parse, the latter function will call the handlers for each relevant element that it finds in the given piece of document. These handlers will need to access the callback table, so lxp_parse puts this table at stack index three (right after the parameters). There is one more detail in the call to XML_Parse: remember that the last argument to this function tells Expat whether the given piece of text is the last one. When we call parse without an argument, s will be NULL, so this last argument will be true.

Now let us turn our attention to the functions f_CharData, f_StartElement, and f_EndElement, which handle the callbacks. All these three functions have a similar structure: each checks whether the callback table defines a Lua handler for its specific event and, if so, prepares the arguments and then calls this Lua handler.

Let us see first the f_CharData handler, in Figure 32.5, “Handler for character data”.

Figure 32.5. Handler for character data

```
static void f_CharData (void *ud, const char *s, int len) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  /* get handler from callback table */
  lua_getfield(L, 3, "CharacterData");
  if (lua_isnil(L, -1)) {  /* no handler? */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* push the parser ('self') */
  lua_pushlstring(L, s, len);  /* push Char data */
  lua_call(L, 2, 0);  /* call the handler */
}
```

Its code is quite simple. The handler receives a `lxp_userdata` structure as its first argument, due to our call to `XML_SetUserData` when we created the parser. After retrieving the Lua state, the handler can access the callback table at stack index 3, as set by `lxp_parse`, and the parser itself at stack index 1. Then it calls its corresponding handler in Lua (when present), with two arguments: the parser and the character data (a string).

The `f_EndElement` handler is quite similar to `f_CharData`; see Figure 32.6, “Handler for end elements”.

Figure 32.6. Handler for end elements

```
static void f_EndElement (void *ud, const char *name) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_getfield(L, 3, "EndElement");
  if (lua_isnil(L, -1)) {  /* no handler? */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* push the parser ('self') */
  lua_pushstring(L, name);  /* push tag name */
  lua_call(L, 2, 0);  /* call the handler */
}
```

It also calls its corresponding Lua handler with two arguments —the parser and the tag name (again a string, but now null-terminated).

Figure 32.7, “Handler for start elements” shows the last handler, `f_StartElement`.

Figure 32.7. Handler for start elements

```
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_getfield(L, 3, "StartElement");
  if (lua_isnil(L, -1)) {  /* no handler? */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* push the parser ('self') */
  lua_pushstring(L, name);  /* push tag name */

  /* create and fill the attribute table */
  lua_newtable(L);
  for (; *atts; atts += 2) {
    lua_pushstring(L, *(atts + 1));
    lua_setfield(L, -2, *atts);  /* table[*atts] = *(atts+1) */
  }

  lua_call(L, 3, 0);  /* call the handler */
}
```

It calls the Lua handler with three arguments: the parser, the tag name, and a list of attributes. This handler is a little more complex than the others, because it needs to translate the tag's list of attributes into Lua. It uses a quite natural translation, building a table that maps attribute names to their values. For instance, a start tag like

```
<to method="post" priority="high">
```

generates the following table of attributes:

```
{method = "post", priority = "high"}
```

The last method for parsers is `close`, in Figure 32.8, “Method to close an XML parser”.

Figure 32.8. Method to close an XML parser

```
static int lxp_close (lua_State *L) {
  lxp_userdata *xpu =
              (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* free Expat parser (if there is one) */
  if (xpu->parser)
    XML_ParserFree(xpu->parser);
  xpu->parser = NULL;  /* avoids closing it again */
  return 0;
}
```

When we close a parser, we have to free its resources, namely the Expat structure. Remember that, due to occasional errors during its creation, a parser may not have this resource. Notice how we keep the parser in a consistent state as we close it, so there is no problem if we try to close it again or when the garbage collector finalizes it. Actually, we will use exactly this function as the finalizer. This ensures that every parser eventually frees its resources, even if the programmer does not close it.

Figure 32.9, “Initialization code for the `lxp` library” is the final step: it shows `luaopen_lxp`, which opens the library, putting all previous parts together.

Figure 32.9. Initialization code for the `lxp` library

```
static const struct luaL_Reg lxp_meths[] = {
  {"parse", lxp_parse},
  {"close", lxp_close},
  {"__gc", lxp_close},
  {NULL, NULL}
};

static const struct luaL_Reg lxp_funcs[] = {
  {"new", lxp_make_parser},
  {NULL, NULL}
};

int luaopen_lxp (lua_State *L) {
  /* create metatable */
  luaL_newmetatable(L, "Expat");

  /* metatable.__index = metatable */
  lua_pushvalue(L, -1);
  lua_setfield(L, -2, "__index");

  /* register methods */
  luaL_setfuncs(L, lxp_meths, 0);

  /* register functions (only lxp.new) */
  luaL_newlib(L, lxp_funcs);
  return 1;
}
```

We use here the same scheme that we used in the object-oriented Boolean-array example from the section called “Object-Oriented Access”: we create a metatable, make its `__index` field point to itself, and put all the methods inside it. For that, we need a list with the parser methods (`lxp_meths`). We also need a list with the functions of this library (`lxp_funcs`). As is common with object-oriented libraries, this list has a single function, which creates new parsers.

Exercises

Exercise 32.1: Modify the function `dir_iter` in the directory example so that it closes the `DIR` structure as soon as it reaches the end of the traversal. With this change, the program does not need to wait for a garbage collection to release a resource that it knows it will not need anymore.

(When you close the directory, you should set the address stored in the userdata to `NULL`, to signal to the finalizer that the directory is already closed. Also, `dir_iter` will have to check whether the directory is closed before using it.)

Exercise 32.2: In the `lxp` example, we used user values to associate the callback table with the userdata that represents a parser. This choice created a small problem, because what the C callbacks receive is the `lxp_userdata` structure, and that structure does not offer direct access to the table. We solved this problem by storing the callback table at a fixed stack index during the parse of each fragment.

An alternative design would be to associate the callback table with the userdata through references (the section called “The registry”): we create a reference to the callback table and store the reference (an integer) in the `lxp_userdata` structure. Implement this alternative. Do not forget to release the reference when closing the parser.

Chapter 33. Threads and States

Lua does not support true multithreading, that is, preemptive threads sharing memory. There are two reasons for this lack of support. The first reason is that ISO C does not offer it, and so there is no portable way to implement this mechanism in Lua. The second and stronger reason is that we do not think multithreading is a good idea for Lua.

Multithreading was developed for low-level programming. Synchronization mechanisms like semaphores and monitors were proposed in the context of operating systems (and seasoned programmers), not application programs. It is very hard to find and correct bugs related to multithreading, and several of these bugs can lead to security breaches. Moreover, multithreading can lead to performance penalties related to the need of synchronization in some critical parts of a program, such as the memory allocator.

The problems with multithreading arise from the combination of preemption with shared memory, so we can avoid them either using non-preemptive threads or not sharing memory. Lua offers support for both. Lua threads (also known as coroutines) are collaborative, and therefore avoid the problems created by unpredictable thread switching. Lua states share no memory, and therefore form a good base for parallelism in Lua. We will cover both options in this chapter.

Multiple Threads

A *thread* is the essence of a coroutine in Lua. We can think of a coroutine as a thread plus a nice interface, or we can think of a thread as a coroutine with a lower-level API.

From the C API perspective, you may find it useful to think of a thread as a stack—which is what a thread actually is, from an implementation point of view. Each stack keeps information about the pending calls of a thread, plus the parameters and local variables of each call. In other words, a stack has all the information that a thread needs to continue running. So, multiple threads mean multiple independent stacks.

Most functions in Lua's C API operate on a specific stack. How does Lua know which stack to use? When calling `lua_pushnumber`, how do we say where to push the number? The secret is that the type `lua_State`, the first argument to these functions, represents not only a Lua state, but also a thread within that state. (Many people argue that this type should be called `lua_Thread`. Maybe they are right.)

Whenever we create a Lua state, Lua automatically creates a main thread within this state and returns a `lua_State` representing this thread. This main thread is never collected. It is released together with the state, when we close the state with `lua_close`. Programs that do not bother with threads run everything in this main thread.

We can create other threads in a state calling `lua_newthread`:

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

This function pushes the new thread on the stack, as a value of type `"thread"`, and returns a `lua_State` pointer representing this new thread. For instance, consider the following statement:

```
L1 = lua_newthread(L);
```

After running it, we will have two threads, `L1` and `L`, both referring internally to the same Lua state. Each thread has its own stack. The new thread `L1` starts with an empty stack; the old thread `L` has a reference to the new thread on top of its stack:

```
printf("%d\n", lua_gettop(L1));          --> 0
printf("%s\n", luaL_typename(L, -1));    --> thread
```

Except for the main thread, threads are subject to garbage collection, like any other Lua object. When we create a new thread, the value pushed on the stack ensures that the thread is not collected. We should never use a thread that is not properly anchored in the state. (The main thread is internally anchored, so we do not have to worry about it.) Any call to the Lua API may collect a non-anchored thread, even a call using this thread. For instance, consider the following fragment:

```
    lua_State *L1 = lua_newthread (L);
    lua_pop(L, 1);          /* L1 now is garbage for Lua */
    lua_pushstring(L1, "hello");
```

The call to `lua_pushstring` may trigger the garbage collector and collect `L1`, crashing the application, despite the fact that `L1` is in use. To avoid this, always keep a reference to the threads you are using, for instance on the stack of an anchored thread, in the registry, or in a Lua variable.

Once we have a new thread, we can use it like the main thread. We can push to and pop elements from its stack, we can use it to call functions, and the like. For instance, the following code does the call `f(5)` in the new thread and then moves the result to the old thread:

```
lua_getglobal(L1, "f");   /* assume a global function 'f' */
lua_pushinteger(L1, 5);
lua_call(L1, 1, 1);
lua_xmove(L1, L, 1);
```

The function `lua_xmove` moves Lua values between two stacks in the same state. A call like `lua_xmove(F, T, n)` pops `n` elements from the stack `F` and pushes them on `T`.

For these uses, however, we do not need a new thread; we could just use the main thread as well. The main point of using multiple threads is to implement coroutines, so that we can suspend their executions and resume them later. For that, we need the function `lua_resume`:

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

To start running a coroutine, we use `lua_resume` as we use `lua_pcall`: we push the function to be called (which is the coroutine body), push its arguments, and call `lua_resume` passing in `narg` the number of arguments. (The `from` parameter is the thread that is doing the call or `NULL`.) The behavior is also much like `lua_pcall`, with three differences. First, `lua_resume` does not have a parameter for the number of wanted results; it always returns all results from the called function. Second, it does not have a parameter for a message handler; an error does not unwind the stack, so we can inspect the stack after the error. Third, if the running function yields, `lua_resume` returns the code `LUA_YIELD` and leaves the thread in a state that can be resumed later.

When `lua_resume` returns `LUA_YIELD`, the visible part of the thread's stack contains only the values passed to `yield`. A call to `lua_gettop` will return the number of yielded values. To move these values to another thread, we can use `lua_xmove`.

To resume a suspended thread, we call `lua_resume` again. In such calls, Lua assumes that all values on the stack are to be returned by the call to `yield`. For instance, if we do not touch the thread's stack between a return from `lua_resume` and the next resume, `yield` will return exactly the values it yielded.

Typically, we start a coroutine with a Lua function as its body. This Lua function can call other functions, and any of these functions can occasionally yield, terminating the call to `lua_resume`. For instance, assume the following definitions:

```
function foo (x)  coroutine.yield(10, x)  end

function foo1 (x)  foo(x + 1); return 3  end
```

Now, we run this C code:

```
lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "foo1");
lua_pushinteger(L1, 20);
lua_resume(L1, L, 1);
```

The call to `lua_resume` will return `LUA_YIELD`, to signal that the thread yielded. At this point, the `L1` stack has the values given to `yield`:

```
printf("%d\n", lua_gettop(L1));             --> 2
printf("%lld\n", lua_tointeger(L1, 1));     --> 10
printf("%lld\n", lua_tointeger(L1, 2));     --> 21
```

When we resume the thread again, it continues from where it stopped (the call to `yield`). From there, `foo` returns to `foo1`, which in turn returns to `lua_resume`:

```
lua_resume(L1, L, 0);
printf("%d\n", lua_gettop(L1));             --> 1
printf("%lld\n", lua_tointeger(L1, 1));     --> 3
```

This second call to `lua_resume` will return `LUA_OK`, which means a normal return.

A coroutine can also call C functions, which can call back other Lua functions. We have already discussed how to use continuations to allow those Lua functions to yield (the section called “Continuations”). A C function can yield, too. In that case, it also must provide a continuation function to be called when the thread resumes. To yield, a C function must call the following function:

```
int lua_yieldk (lua_State *L, int nresults, int ctx,
                              lua_CFunction k);
```

We should use this function always in a return statement, such as here:

```
static inf myCfunction (lua_State *L) {
  ...
  return lua_yieldk(L, nresults, ctx, k);
}
```

This call immediately suspends the running coroutine. The `nresults` parameter is the number of values on the stack to be returned to the respective `lua_resume`; `ctx` is the context information to be passed to the continuation; and `k` is the continuation function. When the coroutine resumes, the control goes directly to the continuation function `k`. After yielding, `myCfunction` cannot do anything else; it must delegate any further work to its continuation.

Let us see a typical example. Suppose we want to write a function that reads some data, yielding if the data is not available. We may write the function in C like this:[1]

```
int readK (lua_State *L, int status, lua_KContext ctx) {
  (void)status;  (void)ctx;  /* unused parameters */
  if (something_to_read()) {
    lua_pushstring(L, read_some_data());
    return 1;
  }
  else
    return lua_yieldk(L, 0, 0, &readK);
}
```

[1] As I already mentioned, the API for continuations prior to Lua 5.3 is a little different. In particular, the continuation function has only one parameter, the Lua state.

```
int prim_read (lua_State *L) {
  return readK(L, 0, 0);
}
```

In this example, `prim_read` does not need to do any initialization, so it calls directly the continuation function (`readK`). If there is data to read, `readK` reads and returns this data. Otherwise, it yields. When the thread resumes, it calls the continuation function again, which will try again to read some data.

If a C function has nothing else to do after yielding, it can call `lua_yieldk` without a continuation function or use the macro `lua_yield`:

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

After this call, when the thread resumes, control returns to the function that called `myCfunction`.

Lua States

Each call to `luaL_newstate` (or to `lua_newstate`) creates a new Lua state. Different Lua states are completely independent of each other. They share no data at all. This means that no matter what happens inside a Lua state, it cannot corrupt another Lua state. This also means that Lua states cannot communicate directly; we have to use some intervening C code. For instance, given two states `L1` and `L2`, the following command pushes in `L2` the string on the top of the stack in `L1`:

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

Because data must pass through C, Lua states can exchange only types that are representable in C, like strings and numbers. Other types, such as tables, must be serialized to be transferred.

In systems that offer multithreading, an interesting design is to create an independent Lua state for each thread. This design results in threads similar to POSIX processes, where we have concurrency without shared memory. In this section, we will develop a prototype implementation for multithreading following this approach. I will use POSIX threads (`pthreads`) for this implementation. It should not be difficult to port the code to other thread systems, as it uses only basic facilities.

The system we are going to develop is very simple. Its main purpose is to show the use of multiple Lua states in a multithreading context. After we have it up and running, we can add several advanced features on top of it. We will call our library `lproc`. It offers only four functions:

`lproc.start(chunk)` starts a new process to run the given chunk (a string). The library implements a Lua *process* as a C *thread* plus its associated Lua state.

`lproc.send(channel, val1, val2, ...)` sends all given values (which should be strings) to the given channel identified by its name, also a string. (The exercises will ask you to add support for sending other types.)

`lproc.receive(channel)` receives the values sent to the given channel.

`lproc.exit()` finishes a process. Only the main process needs this function. If this process ends without calling `lproc.exit`, the whole program terminates, without waiting for the end of the other processes.

The library identifies channels by strings and uses them to match senders and receivers. A send operation can send any number of string values, which are returned by the matching receive operation. All communication is synchronous: a process sending a message to a channel blocks until there is a process receiving from this channel, while a process receiving from a channel blocks until there is a process sending to it.

Like its interface, the implementation of lproc is also simple. It uses two circular double-linked lists, one for processes waiting to send a message and another for processes waiting to receive a message. It uses a single mutex to control access to these lists. Each process has an associated condition variable. When a process wants to send a message to a channel, it traverses the receiving list looking for a process waiting for that channel. If it finds one, it removes the process from the waiting list, moves the message's values from itself to the found process, and signals the other process. Otherwise, it inserts itself into the sending list and waits on its condition variable. To receive a message, it does a symmetrical operation.

A main element in the implementation is the structure that represents a process:

```
#include <pthread.h>
#include "lua.h"
#include "lauxlib.h"

typedef struct Proc {
  lua_State *L;
  pthread_t thread;
  pthread_cond_t cond;
  const char *channel;
  struct Proc *previous, *next;
} Proc;
```

The first two fields represent the Lua state used by the process and the C thread that runs the process. The third field, cond, is the condition variable that the thread uses to block itself when it has to wait for a matching send/receive. The fourth field stores the channel that the process is waiting, if any. The last two fields, previous and next, are used to link the process structure into a waiting list.

The following code declares the two waiting lists and the associated mutex:

```
static Proc *waitsend = NULL;
static Proc *waitreceive = NULL;

static pthread_mutex_t kernel_access = PTHREAD_MUTEX_INITIALIZER;
```

Each process needs a Proc structure, and it needs to access this structure whenever its script calls send or receive. The only parameter that these functions receive is the process's Lua state; therefore, each process should store its Proc structure inside its Lua state. In our implementation, each state keeps its corresponding Proc structure as a full userdata in the registry, associated with the key "_SELF". The auxiliary function getself retrieves the Proc structure associated with a given state:

```
static Proc *getself (lua_State *L) {
  Proc *p;
  lua_getfield(L, LUA_REGISTRYINDEX, "_SELF");
  p = (Proc *)lua_touserdata(L, -1);
  lua_pop(L, 1);
  return p;
}
```

The next function, movevalues, moves values from a sender process to a receiver process:

```
static void movevalues (lua_State *send, lua_State *rec) {
  int n = lua_gettop(send);
  int i;
  luaL_checkstack(rec, n, "too many results");
  for (i = 2; i <= n; i++)  /* move values to receiver */
    lua_pushstring(rec, lua_tostring(send, i));
}
```

It moves to the receiver all values in the sender stack but the first, which will be the channel. Note that, as we are pushing an arbitrary number of elements, we have to check for stack space.

Figure 33.1, "Function to search for a process waiting for a channel" defines the function `searchmatch`, which traverses a list looking for a process that is waiting for a given channel.

Figure 33.1. Function to search for a process waiting for a channel

```
static Proc *searchmatch (const char *channel, Proc **list) {
  Proc *node;
  /* traverse the list */
  for (node = *list; node != NULL; node = node->next) {
    if (strcmp(channel, node->channel) == 0) {  /* match? */
      /* remove node from the list */
      if (*list == node)  /* is this node the first element? */
        *list = (node->next == node) ? NULL : node->next;
      node->previous->next = node->next;
      node->next->previous = node->previous;
      return node;
    }
  }
  return NULL;  /* no match found */
}
```

If it finds one, it removes the process from the list and returns it; otherwise, the function returns `NULL`.

The last auxiliary function, in Figure 33.2, "Function to add a process to a waiting list", is called when a process cannot find a match.

Figure 33.2. Function to add a process to a waiting list

```
static void waitonlist (lua_State *L, const char *channel,
                                      Proc **list) {
  Proc *p = getself(L);

  /* link itself at the end of the list */
  if (*list == NULL) {  /* empty list? */
    *list = p;
    p->previous = p->next = p;
  }
  else {
    p->previous = (*list)->previous;
    p->next = *list;
    p->previous->next = p->next->previous = p;
  }

  p->channel = channel;  /* waiting channel */

  do {  /* wait on its condition variable */
    pthread_cond_wait(&p->cond, &kernel_access);
  } while (p->channel);
}
```

In this case, the process links itself at the end of the appropriate waiting list and waits until another process matches with it and wakes it up. (The loop around `pthread_cond_wait` handles the spurious wakeups allowed in POSIX threads.) When a process wakes up another, it sets the other process's field `channel` to `NULL`. So, if `p->channel` is not `NULL`, it means that nobody matched process `p`, so it has to keep waiting.

With these auxiliary functions in place, we can write `send` and `receive` (Figure 33.3, “Functions to send and receive messages”).

Figure 33.3. Functions to send and receive messages

```
static int ll_send (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);

  pthread_mutex_lock(&kernel_access);

  p = searchmatch(channel, &waitreceive);

  if (p) {  /* found a matching receiver? */
    movevalues(L, p->L);  /* move values to receiver */
    p->channel = NULL;  /* mark receiver as not waiting */
    pthread_cond_signal(&p->cond);  /* wake it up */
  }
  else
    waitonlist(L, channel, &waitsend);

  pthread_mutex_unlock(&kernel_access);
  return 0;
}

static int ll_receive (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);
  lua_settop(L, 1);

  pthread_mutex_lock(&kernel_access);

  p = searchmatch(channel, &waitsend);

  if (p) {  /* found a matching sender? */
    movevalues(p->L, L);  /* get values from sender */
    p->channel = NULL;  /* mark sender as not waiting */
    pthread_cond_signal(&p->cond);  /* wake it up */
  }
  else
    waitonlist(L, channel, &waitreceive);

  pthread_mutex_unlock(&kernel_access);

  /* return all stack values except the channel */
  return lua_gettop(L) - 1;
}
```

The function `ll_send` starts getting the channel. Then it locks the mutex and searches for a matching receiver. If it finds one, it moves its values to this receiver, marks the receiver as ready, and wakes it up.

Otherwise, it puts itself on wait. When it finishes the operation, it unlocks the mutex and returns with no values to Lua. The function `ll_receive` is similar, but it has to return all received values.

Now let us see how to create new processes. A new process needs a new POSIX thread, and a new thread needs a body to run. We will define this body later; here is its prototype, dictated by `pthreads`:

```
static void *ll_thread (void *arg);
```

To create and run a new process, the system must create a new Lua state, start a new thread, compile the given chunk, call the chunk, and finally free its resources. The original thread does the first three tasks, and the new thread does the rest. (To simplify error handling, the system only starts the new thread after it has successfully compiled the given chunk.)

The function `ll_start` creates a new process (Figure 33.4, “Function to create new processes”).

Figure 33.4. Function to create new processes

```
static int ll_start (lua_State *L) {
  pthread_t thread;
  const char *chunk = luaL_checkstring(L, 1);
  lua_State *L1 = luaL_newstate();

  if (L1 == NULL)
    luaL_error(L, "unable to create new state");

  if (luaL_loadstring(L1, chunk) != 0)
    luaL_error(L, "error in thread body: %s",
                  lua_tostring(L1, -1));

  if (pthread_create(&thread, NULL, ll_thread, L1) != 0)
    luaL_error(L, "unable to create new thread");

  pthread_detach(thread);
  return 0;
}
```

This function creates a new Lua state `L1` and compiles the given chunk in this new state. In case of error, it signals the error to the original state `L`. Then it creates a new thread (using `pthread_create`) with body `ll_thread`, passing the new state `L1` as the argument to the body. The call to `pthread_detach` tells the system that we will not want any final answer from this thread.

The body of each new thread is the function `ll_thread` (Figure 33.5, “Body for new threads”), which receives its corresponding Lua state (created by `ll_start`) with only the precompiled main chunk on the stack.

Figure 33.5. Body for new threads

```
int luaopen_lproc (lua_State *L);

static void *ll_thread (void *arg) {
  lua_State *L = (lua_State *)arg;
  Proc *self;  /* own control block */

  openlibs(L);  /* open standard libraries */
  luaL_requiref(L, "lproc", luaopen_lproc, 1);
  lua_pop(L, 1);  /* remove result from previous call */

  self = (Proc *)lua_newuserdata(L, sizeof(Proc));
  lua_setfield(L, LUA_REGISTRYINDEX, "_SELF");
  self->L = L;
  self->thread = pthread_self();
  self->channel = NULL;
  pthread_cond_init(&self->cond, NULL);

  if (lua_pcall(L, 0, 0, 0) != 0)  /* call main chunk */
    fprintf(stderr, "thread error: %s", lua_tostring(L, -1));

  pthread_cond_destroy(&getself(L)->cond);
  lua_close(L);
  return NULL;
}
```

First, it opens the standard Lua libraries and the `lproc` library. Second, it creates and initializes its own control block. Then, it calls its main chunk. Finally, it destroys its condition variable and closes its Lua state.

Note the use of `luaL_requiref` to open the `lproc` library.[2] This function is somewhat equivalent to `require` but, instead of searching for a loader, it uses the given function (`luaopen_lproc`, in our case) to open the library. After calling the open function, `luaL_requiref` registers the result in the `package.loaded` table, so that future calls to require the library will not try to open it again. With true as its last parameter, it also registers the library in the corresponding global variable (`lproc`, in our case).

Figure 33.6, "Extra functions for the `lproc` module" presents the last functions for the module.

[2] This function was introduced in Lua 5.2.

Figure 33.6. Extra functions for the `lproc` module

```
static int ll_exit (lua_State *L) {
  pthread_exit(NULL);
  return 0;
}

static const struct luaL_Reg ll_funcs[] = {
  {"start", ll_start},
  {"send", ll_send},
  {"receive", ll_receive},
  {"exit", ll_exit},
  {NULL, NULL}
};

int luaopen_lproc (lua_State *L) {
  luaL_newlib(L, ll_funcs);  /* open library */
  return 1;
}
```

Both are quite simple. The function `ll_exit` should be called only by the main process, when it finishes, to avoid the immediate end of the whole program. The function `luaopen_lproc` is a standard function for opening the module.

As I said earlier, this implementation of processes in Lua is a very simple one. There are endless improvements we can make. Here I will briefly discuss some of them.

A first obvious improvement is to change the linear search for a matching channel. A nice alternative is to use a hash table to find a channel and to use independent waiting lists for each channel.

Another improvement relates to the efficiency of process creation. The creation of new Lua states is a lightweight operation. However, the opening of all standard libraries is not that lightweight, and most processes probably will not need all standard libraries. We can avoid the cost of opening a library by using the pre-registration of libraries, as we discussed in the section called “The Function `require`”. With this approach, instead of calling `luaL_requiref` for each standard library, we just put the library opening function into the `package.preload` table. If the process calls `require "lib"`, then—and only then—`require` will call the associated function to open the library. The function `registerlib`, in Figure 33.7, “Registering libraries to be opened on demand”, does this registration.

Figure 33.7. Registering libraries to be opened on demand

```
static void registerlib (lua_State *L, const char *name,
                                       lua_CFunction f) {
  lua_getglobal(L, "package");
  lua_getfield(L, -1, "preload");  /* get 'package.preload' */
  lua_pushcfunction(L, f);
  lua_setfield(L, -2, name);  /* package.preload[name] = f */
  lua_pop(L, 2);  /* pop 'package' and 'preload' tables */
}

static void openlibs (lua_State *L) {
  luaL_requiref(L, "_G", luaopen_base, 1);
  luaL_requiref(L, "package", luaopen_package, 1);
  lua_pop(L, 2);  /* remove results from previous calls */
  registerlib(L, "coroutine", luaopen_coroutine);
  registerlib(L, "table", luaopen_table);
  registerlib(L, "io", luaopen_io);
  registerlib(L, "os", luaopen_os);
  registerlib(L, "string", luaopen_string);
  registerlib(L, "math", luaopen_math);
  registerlib(L, "utf8", luaopen_utf8);
  registerlib(L, "debug", luaopen_debug);
}
```

It is always a good idea to open the basic library. We also need the package library; otherwise, we will not have `require` available to open the other libraries. All other libraries can be optional. Therefore, instead of calling `luaL_openlibs`, we can call our own function `openlibs` (shown also in Figure 33.7, “Registering libraries to be opened on demand”) when opening new states. Whenever a process needs one of these libraries, it requires the library explicitly, and `require` will call the corresponding `luaopen_*` function.

Other improvements involve the communication primitives. For instance, it would be useful to provide limits on how long `lproc.send` and `lproc.receive` should wait for a match. In particular, a zero limit would make these functions non-blocking. With POSIX threads, we could use `pthread_cond_timedwait` to implement this feature.

Exercises

Exercise 33.1: As we saw, if a function calls `lua_yield` (the version with no continuation), control returns to the function that called it when the thread resumes. What values does the calling function receive as results from that call?

Exercise 33.2: Modify the `lproc` library so that it can send and receive other primitive types such as Booleans and numbers without converting them to strings. (Hint: you only have to modify the function `movevalues`.)

Exercise 33.3: Modify the `lproc` library so that it can send and receive tables. (Hint: you can traverse the original table building a copy in the receiving state.)

Exercise 33.4: Implement in the `lproc` library a non-blocking `send` operation.