Chapter 6 - Integration: Using Existing Scripting Systems
Game Scripting Mastery
by Alex Varanese
Premier Press © 2003

Previous Next

Lua (And Basic Scripting Concepts)

The first stop on your scripting language tour is the quaint little town of Lua. Lua is a simple, easy-to-use language and scripting system designed to extend any sort of program by giving it the capability to load and execute optionally compiled scripts (which, really, is the goal of virtually any scripting system). Lua the language is paradoxically characterized by both its basic and straightforward syntax, as well its understated but powerful capability to be expanded significantly by the only non-primitive data structure it supports, the *table*. Don't let its mild-mannered appearance fool you, however; Lua's been put to good use in such commercial games as *MDK2* and *Balder's Gate.* It can definitely pull its weight when it has to. Lua the scripting system is equally clean and easy to use; it comes as a single static library coded in pure C and ready to be dropped into any host application for some hot, steamy scripting action.

Before getting into the details of how to write scripts in the Lua language, have a look at the components that the Lua system provides.

The Lua System at a Glance

I think the real beauty of the Lua scripting system is its simplicity. When you initially download the package, you won't find billions of scattered files and executables. Instead, you'll find the include files and libraries needed to link Lua into your host application, as well as a small handful of utilities. That's all you need, and that's all you get. Of course, you can find Lua on the included CD under the `Scripting Systems/Lua/` directory.

The Lua Library

The Lua library is composed mainly of two files: `lua.lib` and `lua.h`. The library in most respects follows the archetypical outline in that it provides a clean API for initializing itself and shutting down, as well as functions for loading scripts, executing them, and building the function call interface that will let them talk back and forth with your host application. I'll get back to the details of how to use this library later.

The `luac` Compiler

Lua comes with an easy-to-use command-line driven compiler called `luac`. Typing `luac` at the command prompt will display the program's usage info. To compile a script, simply type:

```
luac <Filename>
```

where *`Filename`* is the name of the script. The script will be compiled into a file called `luac.out` by default, but this can be changed with the `-o` switch. For example, if you have a script called `test.lua` that you want compiled to a file with the same name, you type this:

```
luac -o test.out test.lua
```

What may surprise you about all this, however, is that you don't ever actually need to use the `luac` compiler in order to use the scripting system. Scripts written in Lua can be loaded directly by the Lua library and will be compiled on-the-fly, at the time they're loaded. This is a nice feature because it allows you to immediately see the results of your script code; you don't have to waste any time on an intermediate compiling step, and you don't have to manage two filenames. The downsides, however, include the fact that you won't get particularly meaningful compile-time errors when your compiling is done at runtime. Because your game (or whatever the host application may be) will be in control of the screen at the time, Lua won't be able to print out a list of syntax errors, for example. The other problem is that loading scripts will now be somewhat slower, as Lua will have to spend the extra time compiling it then and there.

So, `luac` is generally a good program to have around. Not only does it let you compile your scripts ahead of time

for much faster loading at runtime, but it also provides you with the same level of compile-time error information that you'd expect from any other compiler. Another advantage is that you won't have to distribute the source to your scripts with your game; instead, you can just release the compiled binaries, which aren't particularly easy for malicious gamers to hack, and also take up less space. In other words, you don't *have* to use the compiler, but you will most likely want to (and definitely should anyway).

The `lua` Interactive Interpreter

Another utility that comes with Lua is the interactive interpreter. This useful little program, also accessible from the command prompt, simply displays the following upon invocation:

```
>
```

Although the interface is about as friendly as the infamous `DEBUG` utility that ships with MS-DOS, the program lets you immediately test out blocks of Lua code by typing them directly into the interpreter and seeing the results in real time (hence the "interactivity"). I haven't discussed the syntax of Lua yet, but the following should be pretty self-explanatory. For example, if you were to type the following:

```
> X = 32
> Y = 64
> print ( X + Y )
```

You'd see the following output:

```
96
```

The last piece of information regarding the `lua` interactive interpreter worth mentioning is that it can also be used to immediately run simple scripts without the need to embed the `lua.lib` runtime environment into a C program. Simply call `lua` with a filename as the single command-line parameter, like so:

```
lua my_script.lua
```

and it will attempt to execute and print the output of the script. In addition, `lua` will provide the same level of detail in compile-time errors as `luac` will, which can be useful. Lastly, scripts running inside the `lua` interpreter are automatically given a special `print ()` function, which can be used to print values to the screen, much like `printf ()` in C. Even though I haven't discussed Lua syntax yet, the following should be pretty self-explanatory:

```
print ( "Hello, world!" );
```

Running this in `lua`, strangely enough, produces the following output:

```
Hello, world!
```

Keep this function in mind as you read through the following sections.

> **Tip** You'll notice that the interpreter seems to evaluate your statements as soon as you press Enter, even if they're supposed to be part of a larger construct such as an if block. To enter a full block of code without immediately executing it as it's typed, simply follow each line in the block with a backslash (\), much like a multi-line #define macro in C.All of the code will be executed at once after the first non-backslash-terminated line is entered.

The Lua Language

Lua as a language is simple and straightforward. It won't take long to learn the syntax and semantics behind it, and once you have them down, you'll find it elegant and easy to use. The syntax somewhat resembles a mixture of C, BASIC, and Pascal, resulting in a no-frills look and feel that, although not a perfect C clone, should still be an easy transition to make when switching from game engine code to script code. This chapter refers to Lua 4.0, the latest official release at the time of this writing.

The interactive interpreter I mentioned in the last section will be extremely useful during the next few pages; if you really want to follow along, start it up and play with some of the language examples that are discussed. It's the best and fastest way to get really familiar with how Lua works. I highly recommend it.

Comments

I like to introduce comment syntax first when describing a language, because it generally shows up in the code examples anyway. Lua's single comment type is denoted with a double-dash:

```
-- This is a comment.
```

Just like the // comment in C++, Lua's comments cause everything from the double-dashes to the end of the line to be ignored by the compiler. Lua has no provisions for block comments, so multi-line comments must be broken into single lines manually:

```
-- This is the first line of a comment,
-- which is continued down here,
-- and finished here.
```

It's a bit of a hassle, but oh well. :)

Variables

Like most scripting languages, Lua is *typeless*. This means that any variable can hold any value of any type at any time, as opposed to languages like C, which force you to declare a variable of a given type and stick to that type throughout the variable's lifespan. Also unlike C, Lua variables need not be officially declared. Rather, a variable is brought into existence at the time of its first assignment. However, as you'll see, this initial assignment is restricted to some extent in many cases and is often considered a somewhat "implicit" declaration. More on this later.

Identifiers in Lua follow the same rules that exist in C—valid identifiers are sequences of letters, numbers, and underscores that begin with a non-numeric character (meaning a letter or underscore). Identifiers are also case-sensitive, so `myvar`, `myVar`, `MyVar`, and `MYVAR` are all considered different variable names.

Because variables need only be assigned to be declared, the following block of code would declare and initialize two variables, `X` and `Y`:

```
X = 4096            -- Declare X and set its value to 4096
Y = "Hello, world!" -- Declare Y as a string containing "Hello, world!"
```

Caution Avoid creating identifiers that consist of an underscore followed by an all- caps string, such as `_IDENTIFIER`.This convention is used internally by Lua for its own use, and the possibility of a future version of the language defining the same identifier you've used in your scripts may potentially break your code. Besides, they're ugly anyway.

This little example also illustrates another quirk of Lua's syntax: that semicolons aren't required to terminate lines. However, the semicolon can still be used and is still required in the case of statements that span multiple lines. Consider the following:

```
MyVar0 = 128     -- Valid statement; semicolons are optional.
MyVar1 = 256;      -- Also valid; semicolons can be used if preferred.

print (
"This is a long line!"
);            -- Valid, multi-line statements are allowed as long
              -- as the semicolon is present.

print (
"So is this!"
)             -- Invalid, multi-line statements must end with ';'.
```

Even though variables only need to be assigned to be declared, they still can't actually be used as arithmetic expressions without being given some sort of initial value. This is because all variables are assigned `nil` before their first assignment, which doesn't make sense in the case of math operations. For example:

```
U = 1024;
V = 2048;
print ( U + V );
print ( U + V + W );
```

This would produce the following:

```
3072
error: attempt to perform arithmetic on global 'W' (a nil value)
stack traceback:
       1:       main of string "print ( U + V ); ..." at line 4
```

The first line of the output is the sum 3072, just like you would expect, but the following lines are an error message letting you know that W cannot be used to perform arithmetic. I'll discuss nil in more detail in the following section.

Tip Even though it's optional in most cases, I suggest using semicolons to terminate all statements in Lua anyway. Not only does it make the language seem that much more C/C++ like, but it also makes your code clearer and more robust. If you find that a given statement is getting too long and want to break it into multiple lines, having a semicolon already in place will make sure you don't forget to add it afterwards and wind up with a compile-time error. It's just a good rule of thumb to stick with.As a C and/or C++ programmer, it will be a reflex anyway.

The last issue of variables to cover now is the concept of *multiple assignment*, which Lua supports. Multiple assignment allows you to put more than one variable on the left side of the assignment operator, like so:

```
X, Y, Z = 2, 4, 8;
```

After this line executes, X will equal 2, Y will equal 4, and Z will equal 8. This left-to-right order allows you to tell which identifier will receive which value. Multiple assignment works for any sort of assignment, so you can use it to move the value of one set of variables into another as well:

```
U, V, W = X, Y, Z;
Print ( U, V, W );
```

Which will produce the following (assuming you're using the same X, Y, and Z you initialized in the last example):

```
2       4       8
```

If you're anything like me, the first thought you had when you saw this form of assignment notation was "what happens if you don't provide an equal number of variables and values on both sides of the assignment operator?" Fortunately, in another example of Lua's robust nature, this is handled automatically. In the first case, if you don't provide enough values on the right side to assign to all of the variables left side, the extra variables will be assigned nil:

```
X, Y, Z = 16, 32;
```

This will assign X 16 and Y 32, but Z will be set to nil. This even works in cases when the extra variable has already been initialized. For example:

```
U, V, W = 256, 512, 1024;
print ( U, V, W );
U, V, W = 2048,
4096; print ( U, V, W );
```

Even though W was assigned a value in the first assignment, which will be visible in the output of the first print () call, the second assignment will replace it with nil:

```
256     512     1024
2048    4096    nil
```

In the second case, where there aren't enough variables on the right side to receive all of the values on the left, the unused values will simply be ignored, so a line like this:

```
X, Y = 8192, 16384, 32768, 65536;
```

is perfectly legal and will only assign X and Y the first two values. The last two variables will simply vanish without a trace, much like Paulie Shore's career.

Overall, multiple assignment is a convenient shorthand but definitely has potential to make your code less-than-readable. Only use it in cases when you're sure that the code is clearly understandable, and try not to do it for too many variables at once. Don't try to get cute and impress your friends with huge tangles of multiple assignment; it will only result in error-prone code. One good use of the technique; however, is swapping two values in one line easily:

```
X = 16;                    -- Declare some variables
Y = 32;
print ( "Unswapped:", X, Y ); -- Print them out
X, Y = Y, X;               -- Swap them with multiple assignment
print ( "Swapped:", X, Y );   -- Print the swapped values
```

This will produce the following:

```
Unswapped:    16          32
Swapped:      32          16
```

Data Types

Now that you can declare and use variables, you're probably interested in knowing what you can stuff into them. Lua supports six data types:

- **Numeric.** Integer and floating-point values. Unlike C, these two types of numeric values are considered the same data type.
- **String.** A string of characters.
- **Function.** A reference to a formally declared function, much like a function pointer in C (but simpler to use and more discreet).
- **Table.** Lua's most complex and powerful data type; tables can be as simple as associative arrays and as complex as the basis for more advanced data structures like linked lists and classes.
- **Userdata.** A slightly more obscure data type that allows C pointers to be stored in Lua variables for a more tight integration into the host application. Userdata pointers correspond to the `void *` pointer type in C. I won't be covering this data type.
- **nil.** The simplest data type by far, `nil`'s only job is to be different from every other value the language supports. This means it makes a good flag value, especially when you want to mark something as uninitialized or invalid. In fact, any reference to a variable that hasn't been directly assigned a value will equal `nil`. `nil` is also the only concept of "false- hood" the language supports. In other words, `nil` is like a more robust version of C's `NULL`. This is consistent with what you saw in the last section when you tried adding a `nil` value to two integers, which is illegal in Lua. This is an important lesson: `nil` is *false*, but it is *not* equal to zero in a numeric or arithmetic sense. This is why arithmetic expressions involving `nil` variables don't make sense and result in a runtime error.

If you happen to have the Lua interpreter open at the time, try using the `type ()` function to examine various identifiers. The `type ()` function returns a string describing the data type of whatever identifier is passed to it, so consider the following:

```
print ( type ( 256 ) ); \
print ( type ( 3.14159 ) ); \
print ( type ( "It's a trap!" ) );
```

Upon pressing Enter, you should see the following output:

```
number
number
string
```

Right off the bat, the numeric and string types should be a snap, and even the function type is pretty simple when you think about it. `nil` is easy to grasp as well, and the `Userdata` type is beyond the scope of this book so I won't be discussing it any further. That leaves you with tables, which is good because they deserve the most explanation.

Before moving on, however, I'd just like to quickly mention one last aspect of Lua's data types: *coercion*. Coercion is when one data type is cast, or *coerced* into another for the sake of executing an expression. For example, numeric values and strings can be used interchangeably in a number of expressions, like so:

```
print ( 16 + 32 );
print ( "16" + 32 );
print ( 16 + "32" );
print ( "16" + "32" );
```

Each of these `print ()` calls will output the numeric value `48`. This is because whenever a string was encountered in the arithmetic expression, it was coerced into its numeric form. Lua recognizes strings that can be converted meaningfully to numbers, like the previous ones. However, the following statement would cause an error:

```
print ( 16 + "32" + "Alex" );
```

Note Although I'm sure you've picked up on this already, I'd just like to make sure that you're clear on the

`print ()` function. `print ()` will print any value passed to it, as well as the contents of any identifier. This is a special function built in to the version of Lua running in the interpreter to allow immediate feedback while coding interactively.The function also allows you to pass it comma-delimited lists, the output of which will be aligned with tab stops.You'll see more of this later.

The first two values, `16` and `"32"`, are valid. `16` is already an integer value and `"32"` can be coerced into one and still make sense. When the last string value (`"Alex"`) is reached, however, Lua will attempt to convert it to a number and find that it has no numeric equivalent, thus stopping execution to report the error of attempting to use a string in an arithmetic expression:

```
error: attempt to perform arithmetic on a string value
```

Tables

Tables in Lua are, first and foremost, associative arrays not unlike the ones found in other scripting languages like Perl and PHP. Associative arrays are also comparable to the hash table structure provided in the standard libraries for languages like Java and C++.

Tables are indexed with the same syntax as a C array, and are initialized in much the same way. For example, consider the following table declarations that mimic C string and integer arrays:

```
IntArray = { 16, 32, 64, 128 };
StringArray = { "Aho", "Sethi", "Ullman" };
```

Although you didn't have to specify a data type for the table, or even its size, you do use the traditional C-style `{ … }` notation for initialization. Once the tables have their values, they can be accessed much like you'd expect, but with one major difference: the initialized values start at index 1, not zero:

```
print ( IntArray [ 1 ] );
print ( StringArray [ 2 ] );
```

This code will produce the following output:

```
16
Sethi
```

Of course, even though an initialization set is automatically indexed from 1, it doesn't mean index zero can't be used:

```
IntArray [ 0 ] = 8;
print ( IntArray [ 0 ], IntArray [ 1 ], IntArray [ 2 ] );
```

will produce the following output:

```
8       16      32
```

Although it's important to note that index zero is perfectly valid as long as you manually give it a value, the real lesson in the preceding example is your ability to add new elements to a table whenever you need to. Notice that the set of values that initialized the table included only indexes 1 through 4, but you can still expand the array to cover 0 through 4 by simply assigning a value to the desired index. Lua will automatically expand the array to accommodate the new values. In fact, virtually any index you can imagine will already be accessible the moment you create a new table. For example:

```
print ( IntArray [ 0 ] );
print ( IntArray [ 2 ] );
print ( IntArray [ 24 ] );
print ( IntArray [ 512 ] );
```

Even though indexes 24 and 512 are far from the initialization set, check out the output:

```
8
32
nil
nil
```

Neat, huh? Lua automatically created and initialized indexes 24 and 512, allowing you to access them without any sort of out-of-bounds or access-violation errors. In this regard, table indexes are much like typical Lua variables in that they are created only when they are first assigned (or when you initialize them with the `{ … }` notation), but will contain `nil` until then.

The next important aspect of Lua tables is that they are *heterogeneous*, which means that not all indexes must contain the same type of value. For example:

```
MyTable [ 0 ] = 256;          -- Assign an integer to index 0
MyTable [ 1 ] = 3.14159;      -- Assign a float to index 1
MyTable [ 2 ] = "Yahtzee!";   -- Assign a string to index 2
```

The three indexes of this table contain three different data types, further illustrating a table's flexibility. In addition to being able to hold any sort of primitive value, table indexes can also hold references to other tables, which opens the door to endless possibilities. Most obviously, this lets you simulate multi-dimensional arrays, like so:

```
MultiTable = {};
MultiTable [ 0 ] = { "ABC", "DEF", "GHI" };
MultiTable [ 1 ] = { "JKL", "MNO", "PQR" };
MultiTable [ 2 ] = { "STU", "VWX", "YZ" };
print ( MultiTable [ 0 ][ 1 ] );
print ( MultiTable [ 1 ][ 2 ] );
print ( MultiTable [ 2 ][ 3 ] );
```

Which will output the following:

```
ABC
MNO
YZ
```

It's important to know exactly how things are working under the hood when working with tables that contain tables, however. When working with Lua, don't think of tables as values, but rather as *references*. Any time you access a table index or assign a table to another table index, you're actually dealing with the references Lua maintains for these tables, *not* the values themselves. For example, the output of the following code snippet could represent some serious logic errors if you aren't aware of what's happening:

```
X = {};                           -- Declare a table
X [ 0 ] = 16;                     -- Give it three indexes
X [ 1 ] = 32;
X [ 2 ] = 64;
print ( "X: ", X [ 1 ] );         -- Print out index 1
Y = {};                           -- Declare a new table
Y [ 0 ] = X;                      -- Give it one index, containing X
Y [ 0 ][ 1 ] = "String";          -- Set the index 1 of index 0 to a string
print ( "Y: ", Y [ 0 ][ 1 ] );    -- Print out index 1 of index 0 of Y
print ( "X: ", X [ 1 ] );         -- Print out index 1 of X
```

As you can see, the assigning of X to `Y [ 0 ]` didn't copy the X table and all of its values. Rather, `Y [ 0 ]` was simply given a *reference* to X, which means that any subsequent changes made to the table located at `Y [ 0 ]` will also affect X, as can be seen in the output. This is a lot like pointers in C, but I'll keep the pointer analogies to a minium because this topic can be confusing enough as it is. Refer to Figure 6.8 for an illustration

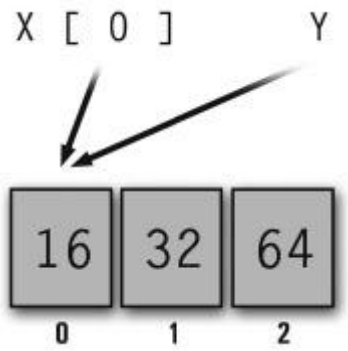

Figure 6.8: Both X and Y are refer ring to the same physi cal data; as a result, any changes to either reference will appear to affect the other.

Moving on, the next major aspect of Lua tables to discuss is their associative nature. In other words, instead of being forced to use integer indexes to index your array, you can use values of any type. In this regard, tables

work on the principal of *key : value pairs*, which let you associate values with other values, called keys, for more intuitive indexing. Consider the following example:

> **Note** Even though I indexed `MutliTable []` from 0 to 2, each of the other three-index tables that were directly initialized at `MultiTable [ 0 ]`, `MultiTable [ 1 ]`, and so on, are indexed automatically 1 to 3 because of Lua's one-index convention. I automatically use zero-indexing out of habit, but it's definitely important to keep Lua's style in mind. Forgetting this detail can lead to some nasty logic errors.

```
Enemy = {};
Enemy [ "Name" ] = "Security Droid";
Enemy [ "HP" ] = 200;
Enemy [ "Weapon" ] = "Pulse Cannon";
Enemy [ "Sprite" ] = "../gfx/enemies/security_droid.bmp";
print ( "Enemy Profile:" );
print ( "\n  Type:", Enemy [ "Name" ],
        "\n    HP:", Enemy [ "HP" ],
        "\nWeapon:", Enemy [ "Weapon" ] );
Enemy Profile:

  Type: Security Droid
    HP: 200
Weapon: Pulse Cannon
```

Which will print out the following:

```
Enemy Profile:

   Type: Security Droid
      HP: 200
Weapon: Pulse Cannon
```

As you can see, each of table's elements was indexed with strings as opposed to numbers. To use the previous terminology, `"Name"`, `"HP"`, `"Weapon"`, and `"Sprite"` were the table's *keys*. The keys were associated with values, which appeared on the right side of the assignment operator. For instance, `"Name"` was the key to the value `"Security Droid"`. This example also introduced you to the `\n` escape code for newlines, which functions just as it does in C. You'll see the rest of Lua's escape codes later.

Any literal data type can be used as a key, so integers, floating-point values, and of course strings, are all valid. Lua also provides an extra notational convenience for instances where the string key is also a valid identifier. For example, consider the following rewrite of the previous example:

```
Enemy = {};
Enemy.Name = "Security Droid";
Enemy.HP = 200;
Enemy.Weapon = "Pulse Cannon";
Enemy.Sprite = "../gfx/enemies/security_droid.bmp";
print ( "Enemy Profile:" );
print ( "\n  Type:", Enemy.Name,
   "\n    HP:", Enemy.HP,
   "\nWeapon:", Enemy.Weapon );
```

As you can see, the string keys are now being used as if they were fields of a `struct`-like structure. In this case, that's exactly what they are. Lua automatically adds these identifiers to the table, allowing them to be accessed in this way. This technique is completely interchangeable with string keys, so the following code:

```
Table = {};
Table.X = 16;
Table [ "Y" ] = 32;
print ( Table [ "X" ], Table.Y );
```

will output:

```
16    32
```

as if everything was declared using the same method. Internally, Lua doesn't care, so `Table [ "Key" ]` is always equivalent to `Table.Key`, provided that `"Key"` is a string containing a valid identifier.

Advanced String Features

You've seen how basic string syntax works in Lua, but there are a few slightly more advanced topics worth covering before moving on. The first is *escape sequences*, which are special character codes preceded by a backslash (\) and direct the compiler to replace certain parts of the string before compilation instead of taking them literally. As an example of when escape sequences are necessary, imagine wanting to use a double quote in a string, such as in the following example:

```
Quote = ""Welcome to the real world", she said to me, condescendingly.";
```

The problem is that the compiler will think the string ends immediately after the second double quote (which is really just supposed to denote the beginning of the quotation), which is in reality the first character in the string. Everything following this will be considered erroneous. Escape sequences help you alleviate this problem by giving the compiler a heads-up that certain quotes are not meant to begin or end the string, but are just characters *within* a larger string. The escape sequence \" (backslash-double quote) is used to do just this. With escape sequences, you can rewrite the previous line and compile it without problems:

```
Quote = "\"Welcome to the real world\", she said to me, condescendingly.";
```

There are a number of escape sequences supported by Lua in addition to the previous one, but most are related to text formatting and are therefore not particularly useful when scripting games. However, I personally find the following useful: \\ (Backslash), \' (Single Quote), and \XXX, where XXX is a three-digit decimal value that corresponds to the ASCII code of the character that should replace the escape sequence.

Using the \" escape sequence can be a pain, however, when dealing with strings that contain a lot of double quotes. Because this is a possibility when scripting games (because many scripts will contain heavy amounts of dialog that possibly require double quotes), you may want to avoid the problem altogether by using single-quotes to enclose your strings, which Lua also supports. For example, consider the following:

```
PrintQuote ( 'You run into the room. "No!" you scream, as you notice your gun is
missing.' );
```

The previous string is equivalent to the following line, but easier to write (and more readable):

```
PrintQuote ( "You run into the room. \"No!\" you scream, as you notice your gun is
missing." );
```

Of course, if for some reason you need to use a large number of single quotes, you can just stick to the double-quoted string.

Lastly, Lua supports a third method of enclosing strings that is by far the most powerful. Enclosing your string with double brackets, such as the following line, allows you to insert physical line breaks directly into the string value without causing a compile-time error:

```
MyString = [[This is a
multi-line
string.]];
print ( MyString );
```

This will produce the following output:

```
This is a
multi-line
string.
```

Expressions

Expressions in Lua are a bit more like Pascal than they are like C, in that they offer a more limited set of operators and use text mnemonics for certain operators instead of symbols. Lua's many operators are organized in Tables 6.1 through 6.3.

Table 6.1: Lua Arithmetic Operators

Operator	Function
+	Add
-	Subtract

*	Multiply
/	Divide
^	Exponent
-	Unary negation
..	Concatenate (strings)

Table 6.2: Lua Relational Operators

Operator	Function
==	Equal
~=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Table 6.3: Lua Logical Operators

Operator	Function
and	And
or	Or
not	Not

Major differences from C worth noting are as follows: the != (Not Equal) operator is replaced with the equivalent ~= operator, and the logical operators are now mnemonics instead of symbols (and instead of &&). These are important to remember, as it's easy to forget details like this and have a "C lapse". :)

Conditional Logic

Now that you have a handle on statements, expressions, and values, you can start structuring that code with conditional logic. Like C and indeed most high-level languages, Lua uses the tried-and- true if statement, although its syntax is most similar to BASIC:

```
if <Expression> then
       Block;
elseif <Expression> then
       Block;
end
```

Unlike C, the expression does not have to be enclosed in parentheses, but you can certainly add them if you want. Expressions can contain parentheses even when they aren't necessary. Here's an example of using if:

```
X = 16;
Y = 32;
if X > Y then
       print ( "X is greater." );
else
       print ( "Y is greater." );
end
```

Lua does not support an analog to C's switch construct, so you can instead use a series of elseif clauses to simulate this (and indeed, this is done in C at times as well). For example, imagine you have a variable called Item that keeps track of an item the player is carrying and implements its behavior when used. Normally one might use a switch to handle each possible value, but you have to use an if-elseif-else chain instead.

```
if Item == "Sword" then
       -- Handle sword behavior
elseif Item == "Morning Star" then
       -- Handle morning star behavior
elseif Item == "Nunchaku" then
       -- Handle nunchaku behavior
else
       -- Unknown item
end
```

As you can see, the final `else` clause mimics C's `default` case for `switch` blocks. As a gentle reminder, remember that the logical operators in Lua follow a different syntax from C:

```
X = 1;
Y = nil;
if X ~= Y then
       print ( "X does not equal Y." );
end
if X and Y then
       print ( "Both X and Y are true." );
end
if X or Y then
       print ( "Either X or Y is true." );
end
if not ( X or Y ) then
   print ( "Neither X nor Y is true." );
end
```

Iteration

The last control structures to consider when discussing Lua are its iterative structures (in other words, its loops). Lua supports a number of familiar loop types: `while`, `for`, and `repeat`. `while` and `for` should make C programmers feel at home, and Pascal users will appreciate the inclusion of `repeat`. All of the structures have a fairly predictable syntax, so take a look at all of them:

```
while <Expression> do
       -- Block
end

for <Index> = <Start>, <Stop>, <Step> do
       -- Block
end

repeat
       -- Block
until <expression>
```

That should all look pretty reasonable, although the exact syntax of the `for` loop might be a bit confusing. Unlike C, which allows you to use entire statements (or even multiple statements) to define the loop's starting condition, stopping condition, and iterator, Lua allows only simple numeric values (in this regard, it's a lot like BASIC). The step value is also optional, and omitting it will cause the loop to default to a step of 1. Take a look at some examples:

```
for X = 0, 3 do
        print ( "Iteration:", X );
end
```

This code will produce:

```
Iteration:    0
Iteration:    1
Iteration:    2
Iteration:    3
```

As you can see, the step value was left out and the loop counting from 0 to 3 in steps of 1. Here's an example with the step included:

```
for X = 0, 7, 2 do
        print ( "Iteration:", X );
end
```

It produces:

```
Iteration:    0
Iteration:    2
Iteration:    4
Iteration:    6
```

Before moving on, I should mention an alternative form of the `for` loop that you might find useful. This version is specifically designed for traversing tables, and looks like this:

```
for <Key>, <Value> in <Table> do
        -- Block
end
```

This form of the loop traverses through each key : value pair of `Table`, and sets `Key` and `Value` appropriately at each iteration. `Key` and `Value` can then be accessed within the loop. For example:

```
MyTable = {};
MyTable [ "Key0" ] = "Value0";
MyTable [ "Key1" ] = "Value1";
MyTable [ "Key2" ] = "Value2";
for MyKey, MyValue in MyTable do
       print ( MyKey, MyValue );
end
```

produces the following output:

```
Key0    Value0
Key2    Value2
Key1    Value1
```

Note Notice that in the first example for the table- traversing form of the `for` loop, the values seem to have been printed out of order.The key : value pair `"Key2"`,`"Value2"` came before `"Key1"`,`"Value1"`.This is because associative arrays don't have the same numeric order that integer-indexed tables do, so the order at which elements are added is not necessarily the element in which they are stored.

Functions

Functions in Lua follow a pattern similar to that of most languages, in that they're defined with an initial declaration line, containing an identifier and a parameter list, followed by a code block that implements the function. Here's an example of a simple function that adds two numbers and returns the sum:

```
function Add ( X, Y )
       return X + Y;
end
print ( Add ( 16, 32 ) );
```

The output, of course, is `48`. The only real nuance regarding functions is that unlike most languages, all variables referenced or created in a function are in the global scope by default. So, for example, imagine changing the previous code so that it looks like this:

```
function Add ( X, Y )
       return X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

Now, instead of printing the return value of the `Add ()` function, you print the uninitialized `GlobalVar`. Not surprisingly, the output is simply nil. However, when you add another line:

```
function Add ( X, Y )
       GlobalVar = X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

You once again get the proper output of 48. This is because GlobalVar is automatically created in the global scope, and therefore is visible even after Add () returns. To suppress this and create local variables, the local keyword is used. So, if you simply add one instance of local to the previous example:

```
function Add ( X, Y )
       local GlobalVar = X + Y;
end
Add ( 16, 32 );
print ( GlobalVar );
```

The output of the script is once again nil, as it would be in most other languages. This is because GlobalVar is created only within the Add () function's scope (so you should probably consider renaming it "LocalVar"), and is therefore invisible once it returns.

The last thing to mention about functions is that they too can be assigned to variables and even table elements. Imagine two variables called Add () and Sub (), which each perform their respective arithmetic operation:

```
function Add ( X, Y )
      return X + Y;
end

function Sub ( X, Y )
      return X - Y;
end
```

You could assign either of these functions to a variable called MathOp, like this:

```
MathOp = Add;
```

And could then call the Add () function indirectly by "calling" MathOp instead:

```
print ( MathOp ( 16, 32 ) );
```

The output will be 48. The interesting thing, however, is what happens when all you change is the function that you assign to MathOp:

```
MathOp = Sub;
print ( MathOp ( 16, 32 ) );
```

Because MathOp now refers to the Sub () function, your output will be -16. As mentioned previously, this capability to "assign" functions to variables is like a somewhat simplified version of C's function pointers. Use it wisely, my friend.

One last detail; because functions can be assigned to table elements, you can take advantage of the same notational shorthands. For example:

```
function PrintHello ()
      print ( "Hello, World!" );
end
MyTable = {};
MyTable [ "Greeting" ] = PrintHello;
```

At this point, the "Greeting" element of MyTable contains a reference to PrintHello (), which can now be called in two ways:

```
MyTable [ "Greeting" ] ();
MyTable.Greeting ();
```

Both are valid and considered equivalent as far as Lua is concerned, but I personally prefer the latter version because it looks more natural.

Note Again, if you're anything like me, a gear or two may have started to turn when you saw the last example."Functions? Stored in tables and accessible just like methods in a class? Hmmmm…" Yes, my friends, this is a small part of the puzzle of how Lua can emulate object-orientation. I won't be covering that in this book, but it's certainly an interesting topic to investigate. See if you can figure out the rest!

Integrating Lua with C

Now that you understand the Lua language enough to get around, it's time for the ***real*** fun to begin. In a moment, you'll return to the bouncing alien head demo and recode the majority of its core logic with Lua as an example of true script integration. But before you go that far, you need to first get your feet wet by getting Lua to run inside and interact with a simple console application to make sure you understand the basics.

The first goal is decidedly simple; write one or two basic scripts, load them in a simple console application, and print some basic output to the screen that illustrates the interactions between the C program and Lua.

Specifically, this program illustrates the following techniques:

- Loading Lua script files and executing them.
- Exporting a C function so that it can be called from Lua scripts.
- Importing Lua functions from scripts so that they can be called from C.
- Passing parameters and returning values in a number of data types to and from both C and Lua.
- Reading and writing global variables in Lua scripts.

Compiling a Lua Project

Understanding how to compile a Lua project is the first and most important thing to understand for obvious reasons. Not surprisingly, the first step is to include `lua.h` in your main source file and make sure the compiler knows where to find the `lua.lib` library.

In the case of Microsoft Visual C++ users, this is a simple matter of selecting Options under the Tools menu and activating the Directories tab. Once there, set the Show Directories For pop-up menu to Include Files. Click the new directory button (the document icon with the sparkle in the upper-left corner) and enter the path to your Lua installation folder (which should contain `lua.h`). Next, set the Show Directories For pop-up to Library Files and repeat what you did for the include files (as long as that same directory also includes `lua.lib`). Figure 6.9 shows the Options dialog box.

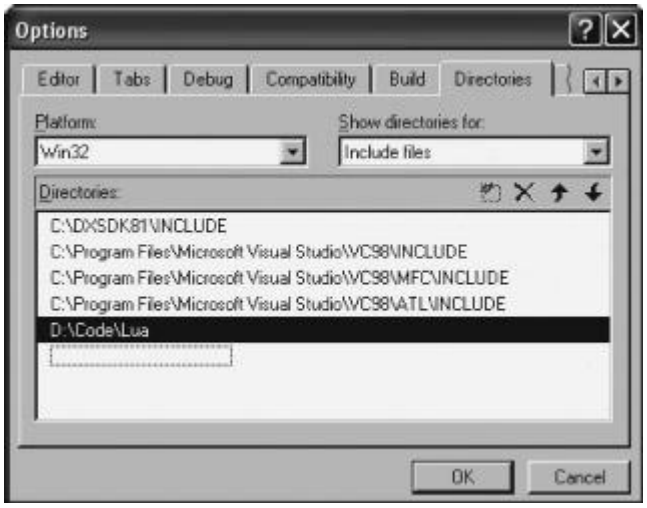

Figure 6.9: The Visual C++ Options dialog box.

Once these settings are complete, make sure to physically include `lua.lib` in your project. I like to put mine under a Libraries folder within the project.

Including the header file is simple enough, but there is one snag. Lua is a *pure-C* library. That may not mean much these days, when popular compilers pretty much blur the difference between C and C++ programs, but unless you're using a pure C programming environment, your linker will have some issues with it if you don't explicitly mention this fact. So, make sure to include `lua.h` like this:

```
extern "C"
{
   #include <lua.h>
```

```
}
```

Remember, this will work only if you properly set your path as described previously.

> **Note** In case you're not familiar with it, `extern` is a *directive* that informs the linker that the identifiers (namely functions) defined within its braces follow the conventions of another language and should be treated as such. In this case, because most people are using the C++ linker that ships with Microsoft Visual C++, you need to make sure it's prepared for a C library that uses slightly different conventions when declaring functions and the like.

Initializing Lua

Lua works on the concept of *states*. A Lua state is essentially a structure that contains information regarding a specific instance of the runtime environment. Each state can contain one script at any time, which is loaded into memory for use. To load and execute multiple scripts concurrently, one needs only to initialize multiple states.

Think about states in the same way you'd think about two instances of the same program in memory. Imagine starting Photoshop (if you don't own Photoshop, imagine owning it as well). Now imagine loading Photoshop again, thus creating two instances of the program at once. Each instance exists in its own "space," and is unrelated to and unaffected by the other. You can open a photo of your dog in one instance, and while doing post-production work on a 3D rendering in the other. Both instances of Photoshop, although essentially the same program with the same functionality, are doing different things at the same time without any knowledge of each other.

From the perspective of the host application, a Lua state is simply a pointer to `lua_State` structure. Once you've declared such a pointer, you can call `lua_open ()` to intialize the state. The only parameter required by `lua_open ()` is the stack size that this particular state will require. Don't worry too much about this; stack size will really only affect the state's ability to handle excessive nesting of function calls, so unless you're going to be hip deep in recursive algorithms, just set it to something like 1024 and forget about it (even this is overkill, but memory is cheap these days so go nuts!). In the relatively unlikely event that you run into stack-overflow errors, just increase it. Here's an example:

```
lua_State * pLuaState = lua_open ( 1024 );
```

> **Note** You can also pass zero to `lua_open ()`, which will cause the stack size to default to 1024 elements.

This example creates a new state called `pLuaState` that refers to an instance of the runtime environment with a stack of 1024 elements. This state is now valid, and is capable of loading and executing scripts.

Of course, no initialization function is complete without its corresponding shut down function. Once you're done with your Lua state, be sure to close it with `lua_close`:

```
lua_close ( lua_State * pLuaState );
```

Loading Scripts

Loading scripts is just as easy as initializing the Lua state. All that's necessary is calling `lua_dofile ()` and passing it the appropriate filename of the script, as well as the state pointer you just initialized. `lua_dofile ()` has the following signature:

```
int lua_dofile ( lua_state * pLuaState, const char * pstrFilename );
```

To execute a script stored in the file "`my_script.lua`", you enter the following:

```
iErrorCode = lua_dofile ( pLuaState, "my_script.lua" );
```

The `pLuaState` instance of the runtime environment will now load, verify, and immediately execute the file. Keep in mind that `lua_dofile ()` will load both compiled and uncompiled scripts transparently; you can pass it either type of file and it will automatically detect and handle it properly. However, because uncompiled scripts will need to be compiled before they can be executed, they will take slightly longer to load. Also, uncompiled scripts are not necessarily valid and may contain syntactic or semantic errors that a compiler would normally not allow. In this case, the call to `lua_dofile ()` will not succeed, so let's discuss its potential error codes. Refer to Table 6.4 for a complete listing.

Once the script is loaded, it is immediately executed. This isn't always what you want; many times, you'll want to load a script ahead of time and execute it later, or even better, execute different parts of it at different times. I'll cover this in a moment. For now, let's just focus on simply loading and running scripts.

Note As you can see, the only shred of compile-time error information `lua_dofile ()` will give you is `LUA_ERRSYNTAX`, which is pretty much one step above nothing at all. Let this be another example of how useful the `luac` compiler is, which gives you a rundown of compile-time errors in detail beforehand. Don't be lazy! Use it!

You can load scripts, but how will you actually know if they're doing anything? You don't have any way to print text from the Lua script to your console application, so even if the script works, you have no way to observe it. This means that even before you write and execute a Lua script,

Table 6.4: `lua_dofile ()` Error Codes

Code	Description
`0`	Success.
`LUA_ERRRUN`	An error occurred while running the script.
`LUA_ERRSYNTAX`	A syntax error was encountered while pre-compiling the script.
`LUA_ERRMEM`	The required memory could not be allocated.
`LUA_ERRERR`	An error occurred with the error alert mechanism. Kind of embarrassing, huh?. :)
`LUA_ERRFILE`	An error occurred while attempting to open or read from the file.

you have to learn how to call C functions from Lua. Once you can do this, you just wrap a function that wraps `printf ()` or something along those lines, and you can print the output of your scripts to the console and actually watch it run.

As such, pretty much everything following this point deals with how Lua and C are integrated, starting with the all-important Lua stack.

The Lua Stack

Lua communicates with C primarily through a stack structure that can be used to pass everything from the values of global variables to function references to parameters to return values. Lua uses this stack internally for a number of tasks, but all you care about is how you can use it to talk to Lua scripts and interpret their responses.

Let's first take a look at some of the generic stack-manipulation functions and macros that Lua provides. It might not make total sense just yet as to how these are used or why, but rest assured it will all make sense soon. You should come to understand the basics of these functions before learning how to apply them.

Much like tables, Lua stacks are indexed starting from 1. This is important to know because the stack does not have to be accessed in a typical stack fashion at all times. The traditional "pushand-pop" stack interface is always available, but you can refer to specific elements of the stack much like you do an array when necessary.

At any time, the index of the stack's top element will be equal to stack's overall size. This is because Lua indexes the stack starting from 1; therefore, a stack of one element can be indexed from 1-1, a stack of 16 elements can be indexed from 1-16, and so on. This is a stark contrast from C and most other languages, in which arrays and other aggregate structures begin indexing from 0. In these cases, the "top" or "last" element in the structure is always equal to the size *minus one*. Figure 6.10 shows you the Lua stack visually.

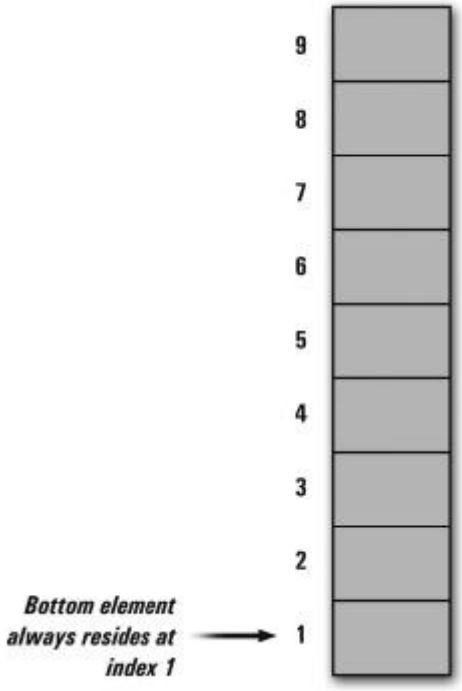

Figure 6.10: The Lua stack.

A program's stack is a turbulent data structure; as functions are called and expressions are evaluated, it grows and shrinks in an erratic pattern. Because of this, stacks are usually accessed in relative terms. For example, when a given function is active, it usually works with its own local portion of the stack, the offset of which is usually passed by the runtime environment.

In the case of Lua, you'll generally be accessing the stack to do one of two things: to write a C function that your scripts can call, or to access your script's global variables. In both cases, the Lua stack will be presented to your program such that the indexes begin at 1. In essence, Lua "protects" the rest of the stack that your program isn't accessing, much like memory-protected operating systems like Windows and Linux protect the memory of your computer from a program if it lies outside of its address space. This makes your job a lot easier, because you can always pretend your chunk of the stack begins at 1. Take a look at Figure 6.11, which illustrates this.

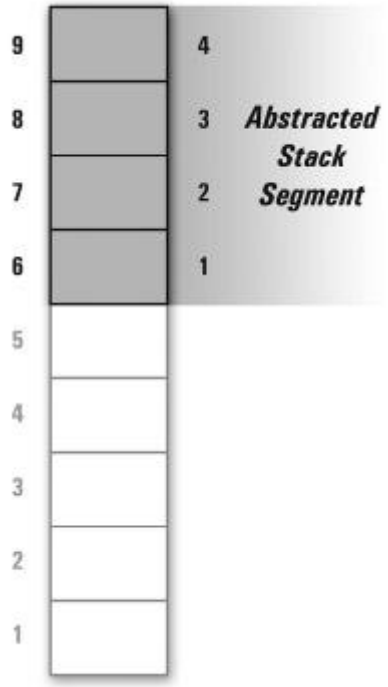

Figure 6.11: Regardless of the size of the stack, Lua will always present what appears to be an empty stack starting from 1 when it is accessed from C.

So to sum things up, Lua will virtually always appear to portray an empty stack starting from 1 when you attempt to access it from C. That being said, let's look at the functions that actually provide the stack interface. Lua features a rich collection of stack-related functions, but the majority of them won't be particularly useful for your purpose and as such, I'll be focusing only on the major ones.

First off, there's `lua_gettop ()`, which gives you the index of the top of the stack:

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

As you learned when you took a look at `lua_open ()`, each Lua state has its own stack size, and thus, its own stack. This means all stack functions (as well as the rest of Lua's functions for that matter) require a pointer to a specific state. Getting back to the topic at hand, this function will return the index of the top element `int`. As you learned, this is also equal to the size of the stack.

Up next is `lua_stackspace ()`, which returns the number of stack elements still available in the stack. So, if the stack size is 1024, and 24 bytes have been used at the time this function is called, 1000 will be returned. This function is especially important because the host application, *not* Lua, is responsible for preventing stack overflow. In other words, if your program is rampantly pushing value after value onto the stack, you run the risk of an overflow error because Lua won't stop or even alert you until it's too late. `lua_stackspace ()` should be used in any case where large numbers of values will be pushed onto the stack, especially when the pushing will be done inside loops, which are especially prone to overflow errors.

The next set of functions you will read about is one of the most important. It provides the classic push/pop interface that stacks are usually associated with. Despite the fact that Lua is typeless, C and C++ certainly aren't, and as such you'll need a number of functions for pushing different data types:

```
void lua_pushnumber ( lua_State * pLuaState, double dValue );
void lua_pushstring ( lua_State * pLuaState, char * pstrValue );
void lua_pushnil ( lua_State * pLuaState );
```

These are three of Lua's `lua_push* ()` functions, but they're the only ones you really have a need for (the rest deal with more obscure, Lua-oriented data types). `lua_pushnumber ()` accepts a double-precision float value, which is a superset of all numeric data types Lua supports (integers, single- and double-precision floating-point). This means that both `int`s and `float`s need to be passed with this function as well. Next is `lua_pushstring ()`, which predictably accepts a single `char *` that points to a typical null-terminated string. The last function worth mentioning is `lua_pushnil ()`, which doesn't require any value, as it simply pushes Lua's `nil` value

onto the stack (which, if you remember, is conceptually similar to C's `NULL`, except that it's not equal to zero).

Popping values off the stack is a somewhat different story. Rather than provide a collection of `lua_pop* ()` functions to match the push functions, Lua simply provides a single macro called `lua_pop ()`, which looks like this:

```
lua_pop ( lua_State * pLuaState, int iElementCount );
```

This macro does nothing more than pops `iElementCount` elements off the stack. They don't actually go anywhere when you pop them, so this function can only be used to remove the values, not extract them. To actually receive the values and store them in C variables, you must use one of the following functions *before* calling `lua_pop ()`:

```
double lua_tonumber ( lua_State * pLuaState, int iIndex );
const char * lua_tostring ( lua_State * pLuaState, int iIndex );
```

Again, the functions should be pretty easy to understand just by looking at them. Give either function an index into the stack, and it will return its value (but will *not* pop or remove that value). In the case of numeric values, you'll always receive a `double` (whether you want an integer or not), and in the case of strings, you'll of course be returned a `char` pointer. Because neither of these functions actually removes the value after returning them, I'll just reiterate that you need to use `lua_pop ()` afterwards if you actually want the value taken off the stack afterwards. Otherwise, these functions can be used to read from anywhere in Lua's stack. To reliably read from the top of the stack every time with these functions, remember to use `lua_gettop ()` to provide the index.

Actually, because Lua doesn't provide a particularly convenient way to directly pop a value off the stack in the traditional context of the stack interface, let's write some macros to do it now. Using the existing Lua functions, you have to do three things in order to simulate a stack pop:

- Get the index of the stack's top element using `lua_gettop ()`.
- Use one of the `lua_to* ()` functions to convert the element at the index returned in the first step to a C variable.
- Use `lua_pop ()` to pop a single element off the top of the stack.

Because this would be a fairly bulky chunk of code to slap into your program every time you want to do this, a nice little macro that wraps this all up into a single call would be great. Here's one that will pop integers off the stack in one fell swoop:

```
#define PopLuaInt( pLuaState, iDest ) \
{                                     \
   iDest = ( int ) lua_tonumber ( pLuaState, lua_gettop
       ( pLuaState ) ); \
   lua_pop ( pLuaState, 1 ); \
}
```

Just pass the macro a valid Lua state and an integer and it will be filled with the proper value. Here's a small code example (assume that `pLuaState` has already been created with `lua_open ()`):

```
int X, Y;
X = 0;
Y = 32;
lua_pushnumber ( pLuaState, Y );
printf ( "X: %d, Y: %d\n", X, Y );
PopLuaInt ( pLuaState, X );
printf ( "X: %d, Y: %d\n", X, Y );
```

The output will be:

```
X: 0, Y: 32
X: 32, Y: 32
```

Try writing similar versions of the macro for floating-point numerics and strings. Be the first kid on your block to collect all three!

So at this point, you can do some basic querying of stack information, and you can push and pop stack values of any data type, as well as perform random access to arbitrary stack indexes (thereby treating it like an array). That's pretty much everything you'll need, but there are a few remaining stack issues to discuss.

First of all, because you now have the ability to read from anywhere in the stack, you should read a bit more about what a valid stack index is. Remember that the Lua stack *always* starts from 1.

Because of this, 0 is never a valid index (unlike tables) and should not be used. Past that, valid indexes run from 1 to the size of the stack. So, if you have a stack of four elements, 1, 2, 3, and 4 are all valid indexes.

One interesting facet of Lua stack access, however, is using a negative number. At first this may seem strange, but using a negative has the effect of accessing the stack "in reverse," so to speak. Index 1 always points to the bottom of the stack, whereas -1 always points to the top. Going back to the example of a four-element stack, consider the following. If index 1 points to the bottom, so does index -4. If index 4 points to the top, so does -1. The same goes for the other elements: element 2 can be indexed with either 2 or -3, whereas element 3 can be accessed with either 3 or -2. Basically, you can always access the stack either relative to the top or relative to the bottom, depending on which is most convenient. Figure 6.12 helps illustrate this concept.

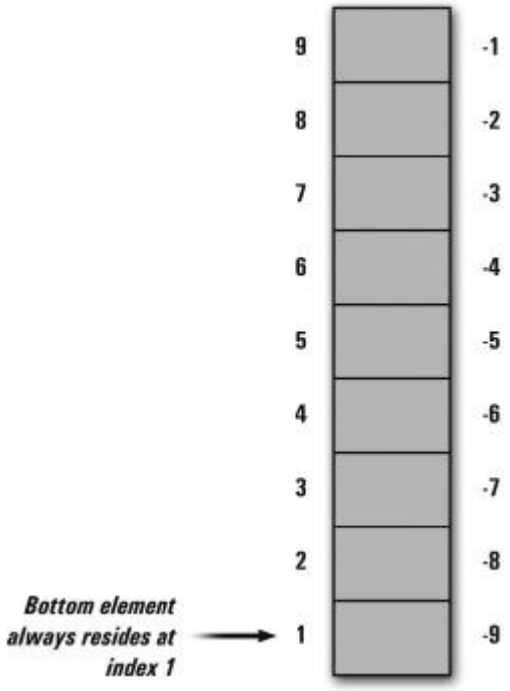

Figure 6.12: Stacks can be accessed relative to either the top or bot tom element, depend ing on the sign of the index. Positive indexes work from the bottom up, whereas negatives work from the top down.

Lastly, let's take a look at a few extra functions Lua provides for determining the type of a given stack element without removing or copying it into a variable first.

```
void lua_type ( lua_State * pLuaState, int iIndex );
void lua_isnil ( lua_State * pLuaState, int iIndex );
void lua_isnumber ( lua_State * pLuaState, int iIndex );
void lua_isstring ( lua_State * pLuaState, int iIndex );
```

The first function, `lua_type ()`, returns one of a number of constants referring to the type of the element at the given index. These constants are shown with a description of their meanings in Table 6.5.

Table 6.5: `lua_type ()` Return Constants

Constant	Description
`LUA_TNIL`	`nil`
`LUA_TNUMBER`	Numeric: `int`, `long`, `float`, or `double`.
`LUA_TSTRING`	String
`LUA_TNONE`	Returned when the specified index is invalid. Nice job, slick!

The other `lua_is* ()` functions work in the same way, but simply return 1 (true) or 0 (false) if the specified index is compatible with the given type. So for example, calling `lua_isnumber ( pLuaState, 8 )`, will return 1 if the element at index 8 is numeric, and 0 otherwise. As you'll learn later in this section, Lua passes parameters to C functions on the stack; when writing a C function that Lua can call, these functions can be useful when attempting to determine whether the parameters passed are of the proper types.

Exporting C Functions to Lua

The process of making a function of the host application callable from Lua (or any scripting system, for that matter) is called *exporting*. To export a function from C to Lua, you simply need to pass a function pointer to the Lua runtime environment, as well as a string containing a name the function should be known by inside the scripts. Lua provides a simple function for this (actu- ally, it's a macro), as follows:

```
lua_register ( lua_State * pLuaState, const char *
pstrFuncName, lua_CFunction pFunc );
```

Given a function name string, the actual function pointer (I'll cover the `lua_CFunction` structure in a second) and the specific Lua state to which this function should be exported, `lua_register ()`, will *register* the function, which allows scripts to refer to it just like any other function. For example, the following script is considered valid if a C function called `CFunc ()` is exported to the state in which it runs:

```
function MyFunc0 ( X, Y )
       -- ...
end
function MyFunc1 ( Z )
       -- ...
end
MyFunc0 ( 16, 32 );
MyFunc1 ( "String Parameter" );
CFunc ( 2, 4.8, "String Parameter" );
```

Of course, if `CFunc ()` is not exported, this will produce a runtime error. Notice, however, that the syntax for calling the C function is identical to any other Lua function, including parameter passing. Speaking of parameters, one detail to remember is that exported C functions do not have well-defined signatures. You can pass any number of parameters of any primitive data type and Lua won't complain. It's the C function's responsibility to sort out the incoming parameters.

To get a feel for how this actually works in practice, let's create that text-printing function discussed earlier, so your subsequent scripts can communicate with you through the console.

The first step, of course, is to write the function. The first attempt at a `printf ()` wrapper might look like this:

```
void PrintString ( char * pstrString )
{
      printf ( pstrString );
      printf ( "\n" );
}
```

This simple wrapper does nothing more than pass `pstrString` to `printf ()` and follow it up with a newline. This is fine as a general-purpose `printf ()` wrapper, but it's not going to work with Lua. Lua requires any C-defined functions to follow a specific function signature, so it can easily maintain a list of function pointers. The prototype of a Lua-compatible C function must look like this:

```
int FuncName ( lua_State * pLuaState );
```

Not only is this signature quite a bit different than the `PrintString ()` wrapper, it looks like it would work only for a function that doesn't require any parameters (aside from the Lua state) and always returns an integer, doesn't it? The reason all functions can follow this same format is because parameters from Lua and return values to Lua are not handled in the same way as they are in C. Both incoming parameters and outgoing results are pushed onto the Lua stack.

Because all incoming parameters are on the stack, you can use Lua's stack interface functions to read them. Remember, at the time your function is called, Lua will make it seem as if the stack is currently empty (whether it is or not), so all of your stack accessing will be relative to element index 1. At the beginning of your C function, the stack will be entirely empty except for any parameters that the Lua caller may have passed. Because of this, the size of the stack is always synonymous with the number of parameters the caller passed, and thus, you can use

`lua_gettop ()`.

Once you know how many parameters have been passed, you can read them using Lua's `lua_to* ()` functions, although you'll need to know what data type you're looking for ahead of time. So, if you wrote a function whose parameter list looked like this:

```
( integer X, float Y, string Z )
```

You could read these three parameters like this:

```
int X = ( int ) lua_tonumber ( pLuaState, 1 );
float Y = lua_tonumber ( pLuaState, 2 );
char * Z = lua_tostring ( pLuaState, 3 );
```

Notice that parameter `X` was at index 1, `Y` was at index 2, and `Z` was at index 3. Lua always pushes its parameters onto the stack in the order they're passed.

Values can be returned in the opposite manner, by pushing them onto the stack before the C function returns. Like passed parameters, return values are pushed onto the stack in the order in which they should be received. Remember, Lua supports multiple assignment and thus multiple return values from functions. If this hypothetical function were to return three more numeric values, the code would look something like this:

```
lua_pushnumber ( pLuaState, 16 );
lua_pushnumber ( pLuaState, 32 );
lua_pushnumber ( pLuaState, 64 );
return 3;
```

Notice that the function returns an integer value corresponding to the number of result values the function should return to Lua (3 in this case). This is very important, as it helps Lua clean up the stack properly afterwards, and can lead to stack corruption errors if this number is not correct. Let's imagine this C function is exported under the name `CFunc ()`. If it's called from Lua in order to return three values, the variables in the following code:

```
U, V, W = CFunc ( X, Y, Z );
```

would be filled in the same order you pushed the values. So, `U` would be set to 16, `V` to 32, and `W` to 64.

> **Tip** Remember, you can always use the `lua_is* ()` functions to validate the data type of the passed parameters. This is especially important because Lua won't force the caller of a host API function to follow a specific prototype, and you have no other way of knowing for sure that the passed parameters are valid.

So you're now capable of registering a C function with Lua, as well as receiving parameters and returning results. That's pretty much everything you need, so let's have a go at implementing that `printf ()` wrapper mentioned earlier. I'll just show you the code up front and I'll dissect it afterwards:

```
int PrintStringList ( lua_State * pLuaState )
{
   // Get the number of strings
   int iStringCount = lua_gettop ( pLuaState );
   // Loop through each string and print it, followed by a newline
   for ( int iCurrStringIndex = 1; iCurrStringIndex <=
      iStringCount; ++ iCurrStringIndex )
   {
      // First make sure that the current parameter on the
      // stack is a string
      if ( ! lua_isstring ( pLuaState, 1 ) )
      {
         // If not, print an error
         lua_error ( pLuaState, "Invalid string." );
      }
      else
      {
   // Otherwise, print a tab, the string, and finally a newline
   printf ( "\t" );
   printf ( lua_tostring ( pLuaState, iCurrStringIndex ) );
   printf ( "\n" );
      }
   }
```

```
   // Return zero, as this function does not return any results
   return 0;
}
```

As you can see the function is now called `PrintStringList ()` and accepts a variable number of string parameters, which are then printed, indented by one tab, and followed by a newline. The function starts with a call to `lua_gettop ()`, which, as you remember, can be used to get the number of parameters when writing host API functions. This value is put in `iStringCount`, and a `for` loop begins in which each string is read from the stack and then printed to the screen. `lua_isstring ()` is used to validate each string. If the parameter is of a non-string type, `lua_error ()` is called. You haven't seen this function before, so I'll take a moment to explain it. Designed for use in console applications, `lua_error ()` accepts a Lua state and a string parameter and halts the current script just before printing the supplied message. Here's the prototype, just for reference:

```
void lua_error ( lua_State * pLuaState, char * pstrMssg );
```

Getting back on track, the rest of the loop deals with reading the string from the stack using `lua_tostring ()` and printing it to the screen (in between the tab and newline characters). The function is finished when the loop ends, and it returns 0 because there were no results to be returned to the Lua caller. Notice also that the parameters passed on the stack are not popped off by the function; this is handled automatically by the Lua runtime environment.

Note When writing host API functions, it helps to be aware that Lua will always ensure that there is at least a minimum number of stack elements avail- able.This number is stored in the `lua.h` constant `LUA_MINSTACK` (which is set to 16, by default).This means that no matter what, your function will always have at least `LUA_MINSTACK` stack elements to work with, although it's always good practice to make sure of this with `lua_stackspace ()`.

Executing Lua Scripts

Now that you have your `PrintStringList ()` written and exported, you're ready to write your first Lua script and watch it execute from within your C host. This first script will be decidedly simple; all you need to do right now is print out a few strings to make sure everything is working right. Once you know you have set everything up correctly, you can accomplish more complex tasks.

This first script will pretty much just do some variable assignment and pass some strings to `PrintStringList ()` to display the results. Let's check it out:

```
-- Create a full name string
FirstName = "Alex";
LastName = "Varanese";
FullName = "Name: " .. FirstName .. " " .. LastName;

-- Now put the floating point value of pi into a string
Pi = 3.14159;
PiString = "Pi: " .. Pi;       -- Numeric values can be automatically coerced to
strings

-- Test some logic
X = 0;                         -- Try setting this to nil instead of zero
if X then
      Logic = "X is true.";    -- Remember, only nil is considered false in Lua
else
      Logic = "X is false.";
end

-- Now call your exported C function for printing the strings
PrintStringList ( "Random Strings:", "" ); -- The extra empty
                                           -- string is just to
                                           -- create a blank line
PrintStringList ( FullName, PiString, Logic );
```

The first part of the script, called `test_0.lua`, creates two string variables, `FirstName` and `LastName`, and uses the `..` string concatenation operator to combine them into `FullName`. The next part uses a floating-point value to create a string containing the first few digits of pi. Notice that Lua automatically casts, or *coerces*, the

floating-point value into a valid string. Next, you create the last string, `Logic`, by setting it to one of two different values depending on whether the variable `X` evaluates to true. This illustrates Lua's definition of truth as any non-`nil` value.

Lastly, with all three strings ready (`FullName`, `PiString`, and `Logic`), you make two calls to `PrintStringList ()` to display them on the console provided by the host C program. Once again, note that the syntax for calling the exported C function was typical Lua syntax, which allows your C functions to blend seamlessly into your Lua-defined functions (even though this script didn't have any).

Returning to the C side of things, your host application's `main ()` function starts with this:

```
// Initialize a Lua state and set the stack size to 1024
lua_State * pLuaState = lua_open ( 1024 );

// Register your simple function with the Lua state for
// printing text strings
lua_register ( pLuaState, "PrintStringList", PrintStringList );

// Print the title
printf ( "Lua Integration Example\n\n" );

// Execute your first test script, which just prints
// random strings
printf ( "Executing Script test_0.lua:\n\n" );
lua_dofile ( pLuaState, "test_0.lua" );
```

All that's necessary to run this script is to initialize Lua with a call to `lua_open ()`, register the `PrintStringList ()` function with `lua_register ()`, and finally load and execute the script in one fell swoop with `lua_dofile ()`. The output of this program will look like this:

```
Lua Integration Example

Executing Script test_0.lua:

       Random Strings:

       Name: Alex Varanese
       Pi: 3.14159
       X is true.
```

Thanks to `PrintStringList ()`, you can be sure that everything went smoothly because the results are right there on the console. Now that you have a simple framework built up for executing Lua, you can try your hand at a more sophisticated example.

Importing Lua Functions

You're probably not too surprised to learn that the opposite of exporting a function from C is importing one from Lua. Naturally, *importing* a function is the process of making that function callable from C, which means that Lua can not only take advantage of C functions you've already written, but your host application can capitalize on any useful functions you may have written in your scripts.

The next script will be primarily focused on demonstrating this concept. To begin, you're going to write a new script, one that defines two functions. The first function will be called `Exponent ()`, and, given two parameters `X` and `Y`, will return `X ^ Y`. The second function, `MultiplyString ()`, will *multiply* a string, which basically just means repeating a string a specified number of times. In other words, `"Hello"` multiplied by four produces the following:

```
HelloHelloHelloHello
```

Although these two functions are indeed simple, they prove educational; between the two of them, they will demonstrate:

- How a Lua function is called from C.
- How both numeric and string parameters are passed to a Lua function from a C host.

- How both numeric and string results can be returned to the C host from Lua functions.

Which is just about everything you need to know about function importing.

Let's get this new script started, which is called test_1.lua, with the `Exponent ()` function:

```
-- Manually computes exponents in the form of X ^ Y
function Exponent ( X, Y )
	-- First, let's just print out the parameters
	PrintStringList ( "Calculating " .. X ..
	     " to the power of " .. Y );
	-- Now manually compute the result
	Exponent = 1;
	if Y < 0 then
		Exponent = -1;	-- Just return -1
					-- for all negative exponents
	elseif Y ~= 0 then
		for Power = 1, Y do
			Exponent = Exponent * X;
		end
	end
	-- Return the final value to C
	return Exponent;
end
```

To make the function more substantial, I've chosen to implement the exponent function with a manual loop that multiplies 1 value by itself Y times. Of course, Lua provides a built-in exponent operator with ^, so there'll be no need for you to do this in practice. Regardless, it works by first setting `Exponent` to 1 and immediately checking for some alternative cases. The first case is a negative power; which isn't supported by the function. Instead, -1 is returned in all such cases. Next, you check to make sure you aren't raising X to the power of zero. If so, you only need to return `Exponent` as is, because raising anything to zero yields 1. Lastly, you handle a valid exponent with the loop described previously. The function concludes with the `return` keyword, which returns the final exponent value to C.

You'll notice I start the function with a call to `PrintStringList ()` that prints a brief message. I do this just to keep some variety going in the C/Lua interaction. Without a simple call to this function, the script would consist entirely of Lua calls, which doesn't illustrate real-world scripting quite as well.

The other function test_1.lua will provide is `MultiplyString ()`:

```
-- "Multiplies" a string; in other words, repeats a string
-- a number of times
function MultiplyString ( String, Factor )
	-- As with the above function, print out the parameters
	PrintStringList ( "Multiplying string \""
	     .. String .. "\" by " .. Factor );
	-- Multiply the string
	NewString = "";
	for X = 1, Factor do
		NewString = NewString .. String;
	end
	-- Return the multiplied string to C
	return NewString;
end
```

This function is even simpler than `Exponent`. All it does is create a variable called `NewString` and assign it the empty string. `NewString` will contain the multipled string and is what you'll return to C. You then enter a simple `for` loop which repeatedly appends `String` to `NewString`, once again using the `..` operator.

With these two functions saved in test_1.lua, you can return to your C host program and add the new code necessary to test it.

The C side of things will get a little more complicated than it's been so far, but it's still nothing you can't handle.

The first thing to understand is that `lua_dofile ()` will no longer immediately execute anything when test_1.lua is loaded. This is because, unlike your previous script, there isn't any code in the global scope. It's like writing a C program without `main ()`. Because all code resides in functions, the Lua runtime environment won't run anything until those functions are called. Because the script never calls any of these functions, in the global scope, nothing ever executes. `lua_dofile ()` has now effectively become a pure script loader, at least conceptually (it will still attempt to run the script, even though nothing will happen).

Once the script is in memory, you can freely call any of its functions at will. Lua doesn't have a particularly high-level mechanism for calling functions, so you'll have to do things fairly manually using the stack. Fortunately, it's still a pretty straightforward process. Have a look.

> **Tip** Remember, you can always optionally compile your scripts. Generally, it's easier to skip the compilation step while you're initially coding and debugging them, but once they're finished, don't forget to run them through `luac`. `lua_dofile ()` is capable of loading both compiled and uncompiled scripts, so you won't have to change your C host (except to change the filename to refer to the compiled version, if it's different). Recall that compiled scripts load faster, are less error-prone, and are much less vulnerable to hacking.

In Lua, functions can be thought of as *globals*, just as much as global variables can be thought of as globals. This doesn't mean they're any more like variables than C functions are, but they can be referred to this way. The first thing you need to do when calling a function is push a reference to the function onto the stack. Because functions are simply another global, you can use `lua_get- global ()` to do the job:

```
lua_getglobal ( pLuaState, "FuncName" );
```

Where `FuncName` is a string value that corresponds to the name of the function within the script. Once the function reference is on the stack, you need to push its parameters on as well. Parameters are pushed onto the stack in left-to-right order. If `FuncName` looks like this:

```
function FuncName ( IntParam, StringParam )
```

And we want to essentially call it like this:

```
FuncName ( 256, "Hello!" );
```

The parameters would be pushed onto the stack like this:

```
lua_pushnumber ( pLuaState, 256 );
lua_pushstring ( pLuaState, "Hello!" );
```

Simple, eh? Now that the function call is represented on the stack in its entirety, you deliver the coup-de-grace by calling `lua_call ()`, which looks like this:

```
lua_call ( lua_State * pLuaState, int ParamCount, int ResultCount );
```

This function will call whatever function was most recently pushed onto the stack, passing `ParamCount` parameters and expecting `ResultCount` results. Remember, due to the multiple assignment capabilities of Lua, functions can return multiple values. If `FuncName ()` accepts the two parameters listed previously and returns one result, the call to `lua_call ()` would look like this:

```
lua_call ( pLuaState, 2, 1 );
```

Lastly, you need to know how to retrieve the result. The result (or results, depending on how many the function returns) will be left on the stack. In your case, assuming `FuncName ()` returned a single integer result, you can use the following code to read it:

```
int iResult = ( int ) lua_tonumber ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
```

You use `lua_tonumber ()` to convert the element at index 1 of the stack to a double-precision floating-point value, and then cast it to an integer to store in the receiving variable. You know the return value is at index 1 because the function only returns one value. The stack is then cleaned up using `lua_pop ()` to remove the return value and bring balance to the force.

That's everything there is to know about basic Lua function calls from the host application. Now that you know what you're doing, let's go back to test_1.lua and try calling your `Exponent ()` and `MultiplyString ()` functions.

```
printf ( "\nLoading Script test_1.lua:\n\n" );
```

```
lua_dofile ( pLuaState, "test_1.lua" );

// Call the exponent function
// Call lua_getglobal () to push the Exponent ()
// function onto the stack
lua_getglobal ( pLuaState, "Exponent" );
// Push two numeric parameters
lua_pushnumber ( pLuaState, 2 );
lua_pushnumber ( pLuaState, 13 );
// Call the function with 2 parameters and 1 result
lua_call ( pLuaState, 2, 1 );
// Pop the numeric result from the stack and print it
int iResult = ( int ) lua_tonumber ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
printf ( "\tResult: %d\n\n", iResult );

// Call the string multiplication function
// Push the MultiplyString () function onto the stack
lua_getglobal ( pLuaState, "MultiplyString" );
// Push a string parameter and the numeric factor
lua_pushstring ( pLuaState, "Location" );
lua_pushnumber ( pLuaState, 3 );
// Call the function with 2 parameters and 1 result
lua_call ( pLuaState, 2, 1 );
// Get the multiplied string and print it
const char * pstrResult;
pstrResult = lua_tostring ( pLuaState, 1 );
lua_pop ( pLuaState, 1 );
printf ( "\tResult: \"%s\"", pstrResult );
```

Everything should pretty much speak for itself; all I've done here is directly applied the technique for calling Lua functions described previously.

At this point, you've learned quite a bit; once you have the ability to call functions from both the host application and the running script, along with parameters and return values, you're pretty much prepared for anything. Most of the interaction between these two entities will lie in function calls. Because you've learned the language as well, you should be familiar enough with Lua in general to get started with your own experiments and exploration. Of course, you still need to get back to the bouncing alien head demo, but before that, there's one last detail of interaction I'd like to show you.

Manipulating Global Lua Variables from C

The last real piece of the C/Lua integration puzzle I'm going to cover is the manipulation of a script's global variables from C. Because globals are often used to control the program on a high level, there are times when you can direct and manipulate the general behavior of your scripts with nothing more than the reading and writing of globals. I personally prefer to keep everything function-based. Rather than directly editing a global variable, I like to assign that global a pair of "setter and getter" functions, which allow me to alter the global's value indirectly and subsequently more safely. However, you're ultimately the one who has to decide how your game's scripts will work, so here's an extra technique for your arsenal in case you personally consider it a better way to go.

As you've seen to some extent, the `lua_getglobal ()` and `lua_setglobal ()` functions can be used to read and write globals indirectly through the stack. Calling `lua_getglobal ()` causes the value of the specified global variable to be pushed onto the stack, whereas `lua_setglobal ()` will pop the value off the top of the stack into the specified global. So, for example, if you wanted to set the value of an integer global called `X`, you simply do the following:

```
lua_pushnumber ( pLuaState, 256 );  -- Push 256 onto the stack
lua_setglobal ( pLuaState, "X" );     -- Move the top stack value into X
```

It's simply a matter of pushing the desired value onto the stack and using lua`_setglobal ()` to move it into place. Likewise, the integer value of `X` could be read with the following code:

```
lua_getglobal ( pLuaState, X );    -- Push X's value onto the stack
int X = ( int ) lua_tonumber ( pLuaState, 1 );  -- Grab the top stack value
```

All you need to do is push the given global's value onto the stack and then convert the value at that index to an integer to store in a C variable. Once again, you're assuming that the stack is empty at the time of the call to `lua_getglobal ()`, which means the value will be placed at index 1. Because this may not always be the case, be sure to use `lua_gettop ()` in practice to get the proper index of the stack's top value. Also, remember to clear the stack off when you're done; calls to `lua_getglobal ()` should generally be followed by a call to `lua_pop ()`.

Let's finish `test_1.lua` by adding some global variables to manipulate. Before the definition of your two functions, let's add the following:

```
GlobalInt = 256;
GlobalFloat = 2.71828;
GlobalString = "I'm an obtuse man...";
```

This gives you three globals to work with, all of differing types. To get things started, let's just try reading their values and printing them from C:

```
// Read some global variables
printf ( "\n\tReading global variables...\n\n" );

// Read an integer global by pushing it onto the stack
lua_getglobal ( pLuaState, "GlobalInt" );
printf ( "\t\tGlobalInt: %d\n", ( int )
   lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Read a float global
lua_getglobal ( pLuaState, "GlobalFloat" );
printf ( "\t\tGlobalFloat: %f\n", lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Read a string global
lua_getglobal ( pLuaState, "GlobalString" );
printf ( "\t\tGlobalString: \"%s\"\n", lua_tostring
   ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
```

Let's expand the example just a bit to write new values to the globals. Of course, you'll re-read them as well to make sure the writes worked:

```
// Write the global variables and re-read them
printf ( "\n\tWriting and re-reading the global variables...\n\n" );

// Write and read the integer global
lua_pushnumber ( pLuaState, 512 );
lua_setglobal ( pLuaState, "GlobalInt" );
lua_getglobal ( pLuaState, "GlobalInt" );
printf ( "\t\tGlobalInt: %d\n", ( int ) lua_tonumber
   ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
// Write and read the float global
lua_pushnumber ( pLuaState, 3.14159 );
lua_setglobal ( pLuaState, "GlobalFloat" );
lua_getglobal ( pLuaState, "GlobalFloat" );
printf ( "\t\tGlobalFloat: %f\n", lua_tonumber ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );

// Write and read the string global
lua_pushstring ( pLuaState, "...so I'll try to be oblique." );
lua_setglobal ( pLuaState, "GlobalString" );
lua_getglobal ( pLuaState, "GlobalString" );
printf ( "\t\tGlobalString: \"%s\"\n", lua_tostring ( pLuaState, 1 ) );
lua_pop ( pLuaState, 1 );
```

Done and done. The last thing to add to your C host is a call to `lua_close ()` to clean everything up:

```
lua_close ( pLuaState );
```

Re-coding the Alien Demo in Lua

Aside from Vader, one last challenge remains. As I mentioned earlier, one of your exercises as you learn each language will be to recode the bouncing alien head demo I showed you at the beginning of the chapter.

Initial Evaluations

As I mentioned earlier, all you really want to do with Lua is set the initial location, velocity, and spin direction of each sprite with the script, as well as produce each frame of the demo by moving the sprites around the screen and handling collisions.

The first thing you need to do is decide exactly what the script will be in charge of. Once you know this, you can establish an appropriate host API— a set of functions that will give the script the capabilities it needs to carry out its tasks.

Because your script will first be responsible for initializing the sprites, let's break down exactly what this entails:

- Set the initial X, Y coordinates to a random on-screen location.
- Set the initial X, Y velocity to random values.
- Set the initial spin direction to a random value (0 or 1).
- Store these values in a script-defined table, just as the original C version stored them in an array.

In short, you need to create a table within the script that will hold all of your bouncing alien heads; each element of the array needs to describe its corresponding alien head in the same way that the `Alien struct` did in the hardcoded version. Obviously, table manipulation is built in to Lua, so you don't need to provide any functionality for that from the host app. What you do need to provide, however, is a function that can generate random numbers.

Once initialization is complete, your script won't be called again until the main loop of the application has begun. Once this takes place, the script will be called once per frame. At each frame, the script will be in charge of the following tasks:

- Blit the background image.
- Loop through each alien in the table and draw it at its current location.
- Blit the completed frame to the screen.
- Update the current frame of animation when the animation timer is active.
- Loop through each alien in the table once again to move it along its current path, and handle collisions as they occur when the movement timer is active.

As you can see, the per-frame part of the script will be required to do a lot more things that Lua isn't directly capable of, so the bulk of your host API will be geared towards these needs. Now that you know what you need, let's lay these functions out.

The Host API

As you've seen, your primary requirements will be generating random numbers, blitting various bitmapped images, and checking the status of timers. With these needs in mind, your host API will look like this:

```
int HAPI_GetRandomNumber ( lua_State * pLuaState );
int HAPI_BlitBG ( lua_State * pLuaState );
int HAPI_BlitSprite ( lua_State * pLuaState );
int HAPI_BlitFrame ( lua_State * pLuaState );
int HAPI_GetTimerState ( lua_State * pLuaState );
```

Notice that I've preceded each of the function names with `HAPI_` (which of course stands for "**H**ost **API**"). This ensures that your host API functions and C-only functions are kept separate. This is just good practice in general when scripting with any language.

As for the functions, they should be fairly self-explanatory, but I'll go over them just in case there's any ambiguity:

- `HAPI_GetRandomNumber ()` accepts two numeric parameters; minimum and maximum values that define a range from which a random number should be chosen and returned to the caller.
- `HAPI_BlitBG ()` is a simple function that causes the background image to be blitted to the framebuffer. No parameters are necessary.
- `HAPI_BlitSprite ()` accepts parameters referring to an X, Y location and an index into the array of frames of the spinning alien head animation.
- `HAPI_BlitFrame ()` is another simple function that blits the framebuffer to the screen. Like `HAPI_BlitBG ()`, no parameters are needed.
- `HAPI_GetTimerState ()` this function accepts a single numeric parameter containing an index that refers to a specific timer. The state of that timer (1 for active, 0 for inactive) is returned to the caller.

With the host API laid out, let's take a look at the new structure of the host application.

The New Host Application

The landscape of the C side of things is quite a bit different now that you're offloading a good portion of the demo's functionality to Lua. Gone is much of the original code, and in its place you find the host API and a number of calls to the Lua system. Speaking of the host API, its one of the biggest changes (or additions, I should say). Have a look at the definitions for a few of the host API functions:

```
int HAPI_GetRandomNumber ( lua_State * pLuaState )
{
	// Read in parameters
	int iMin = GetIntParam ( 1 );
	int iMax = GetIntParam ( 2 );
	// Return a random number between iMin and iMax
	ReturnNumber ( ( rand () % ( iMax + 1 - iMin ) ) + iMin );
	return 1;
}
```

`HAPI_GetRandomNumber ()` does its job in two phases; first the parameters are read in, and then the result is sent out. You start by declaring two integer variables, iMin and iMax, and initialize them with the values returned from `GetIntParam ()`. Wait a second, "`GetIntParam ()`"? What was that?

Throughout the process of rewriting the alien head demo with Lua, there appeared a number of places where macros that wrapped the calls to the actual Lua functions made things a lot cleaner. For example, when a host API function wants to read in an integer parameter, it has to do something like this:

```
int iParam = ( int ) lua_tonumber ( pLuaState, iIndex );
```

First of all, the function `lua_tonumber ()` itself isn't the most intuitive name, at least in this context. What the function is really doing is reading the stack element at `iIndex` and returning it as a numeric value. At least, that's how things are working internally. All you need to worry about, however, is that the function is returning a parameter. So right off the bat, wrapping it in a macro that provides a more descriptive name will result in improved code readability. Second, you have to cast the value the function returns to an `int` because Lua works only with floating-point numerics. Having this cast clog up your code everywhere is just going to make things messier, so the following macro:

```
#define GetIntParam( Index ) \
	( int ) lua_tonumber ( g_pLuaState, Index );
```

just makes everything cleaner, more descriptive, and more concise. This is a trend that you'll find continues throughout this section, so be prepared for a few more macros along these lines.

Where were we? Oh right, `HAPI_GetRandomNumber ()`. Anyway, once you read in the `iMin` and `iMax` parameters, you use another macro, `ReturnNumer ()`, to return the result of a call to the standard C `rand ()` function. `ReturnNumer ()` is very similar to `GetIntParam ()`, except that it of course automates the process of returning a numeric. Let's look at the code:

```
#define ReturnNumer( Num ) \
	lua_pushnumber ( g_pLuaState, Num );
```

Much nicer, eh? Another plus to these macros is that they save you from having to manually pass that Lua state every time you make a Lua call as well. Of course, if you find yourself writing programs that maintain multiple states (which you most likely will, because that's how you implement multiple scripts running at once), you'll lose this luxury.

Overall, `HAPI_GetRandomNumber ()` illustrates an important point when discussing host APIs, because all it really boiled down to was a simple wrapper for `rand ()`. You may find that a large portion of your host API functions don't provide any unique functionality of their own. Rather, they'll usually just wrap existing functions to make the same functions your C program uses accessible to your scripts. Don't worry if you find yourself doing a lot of this. At first it may seem like a lot of extra coding for nothing, but it's the only way to provide your scripts with the functions they're ultimately going to need to be useful.

Let's check out one more host API function, and then I'll move on:

```
int HAPI_BlitSprite ( lua_State * pLuaState )
{
	// Read in parameters
	int iIndex = GetIntParam ( 1 );
	int iX = GetIntParam ( 2 );
	int iY = GetIntParam ( 3 );
	// Blit sprite
	W_BlitImage ( g_AlienAnim [ iIndex ], iX, iY );
	// Return nothing
	return 0;
}
```

Again, you see a similar process. First you read in three integer parameters with your handy `GetIntParam ()` macro. You then pass those parameters directly to the Wrappuh function `W_BlitImage ()`, which performs the blit. Unlike `HAPI_GetRandomNumber (),` this function does not return anything to Lua, hence the return 0.

Moving along, I've created two helper functions for initializing and shutting down Lua in its entirety. `InitLua ()` allows you to open the Lua state and register all of the functions in your host API in one call:

```
void InitLua ()
{
	// Open a new Lua state
	g_pLuaState = lua_open ( LUA_STACK_SIZE );
	// Register your host API with Lua
	lua_register ( g_pLuaState, "GetRandomNumber",
	     HAPI_GetRandomNumber );
	lua_register ( g_pLuaState, "BlitBG", HAPI_BlitBG );
	lua_register ( g_pLuaState, "BlitSprite", HAPI_BlitSprite );
	lua_register ( g_pLuaState, "BlitFrame", HAPI_BlitFrame );
	lua_register ( g_pLuaState, "GetTimerState", HAPI_GetTimerState );
}
```

Notice that the host API functions are not exposed to Lua scripts with the `HAPI_` prefix. I did this because there are so few functions in the script (as you'll soon see), that there's no need to differentiate. Of course, for large script projects you may find it useful to precede your function names with `HAPI_` on both the C and Lua sides of things.

`LUA_STACK_SIZE` is just a constant I've set to 1024. Nothing new.

`InitLua ()` of course has a matching ShutDownLua (), although this function is a bit of a waste, because it only encapsulates one line:

```
void ShutDownLua ()
{
	// Close Lua state
	lua_close ( g_pLuaState );
}
```

What can I say? I'm a bit of a neat-freak, so `InitLua ()` had to have a matching `ShutDown ()` function, whether it was necessary or not. :) It would just seem lopsided without one!

After the call to `InitLua ()`, you'll have a valid Lua state and your host API will be locked and loaded. It's here

where the scripting really begins. After all of your C-side initialization is done, you can initialize your alien head sprites with one call:

```
CallLuaFunc ( "Init", 0, 0 );
```

That's right, another macro has reared its head. This one, aptly entitled `CallLuaFunc ()`, calls Lua functions. (Honestly, sometimes I wish my function names were less descriptive—it makes the explanations of what they mean seem so anticlimactic.) Normally, because a Lua function call involves using `lua_getglobal ()` to put the function reference onto the stack, and then calling `lua_call ()`, this macro helps you out a bit by reducing everything to a single line:

```
#define CallLuaFunc( FuncName, Params, Results ) \
{ \
      lua_getglobal ( g_pLuaState, FuncName ); \
      lua_call ( g_pLuaState, Params, Results ); \
}
```

Just pass it a string containing the function name, the number of parameters, and the number of results.

Anyway, the call to the Lua script was in reference to a function called `Init ()`, as you can see. Because I haven't covered the contents of the script yet, just take this on faith.

Immediately following the call to your script's `Init ()` function, the main loop of the demo begins, which is now rather minimalist because its guts have been transferred to Lua:

```
// Start the main loop
MainLoop
{
       // Start the current loop iteration
       HandleLoop
       {
            // Let Lua handle the frame
            CallLuaFunc ( "HandleFrame", 0, 0 );
            // Check for the Escape key and exit if it's down
            if ( W_GetKeyState ( W_KEY_ESC ) )
                  W_Exit ();
       }
}
```

Another call to `CallLuaFunc ()`, and another script function you haven't yet seen. This one is called HandleFrame (), and naturally, handles the current frame by moving the sprites around. Once again, you'll see these two functions in the next section.

That's it! In summary, the new host application works by first defining a series of functions that collectively form the host API, and then initializes Lua by using `lua_open ()` to create a Lua state and register the host API's functions. At this point, the Lua system is all ready to go, and the script's two functions are called. First Init () is called to initialize the sprites, and `HandleLoop ()` is called once per frame to move them around. Because you're done with the C stuff, you can now move on and actually see these two functions (among other things).

The Lua Script

The Lua script, which I've given the almost frighteningly creative filename `script.lua`, is the only one you'll need for this demo. In it, there are four main elements, as follows:

- An area for declaring constants.
- An area for declaring global variables.
- The first function, `Init ()`.
- The second (and last) function, `HandleFrame ()`.

As you can see, a script is structured in the same way a program is, something you'll discover in more and more depth as your mastery of scripting unfolds. Although scripts and programs are indeed fundamentally and technically different things; they're conceptually the same in most respects.

As I said, your script will consist mostly of a constant declaration section, a global variable declaration section,

and two functions. Notice again that there is no code in the global scope—in other words, code that resides outside the functions—because it would be automatically executed by `lua_dofile ()` and you don't necessarily want anything to be run at that time. Rather, you'd like Lua to sim ply load the file into memory for you and let it sit for you to reference later through function calls when you need to.

> **Tip** Even though this script example has no code in the global scope, and thus no code that automatically runs after the call to `lua_dofile ()`, this isn't always something to avoid. If your script has a block of initialization code that you know you're only going to call once at the time the script is loaded, you might as well put this code in the global scope so `lua_dofile ()` automatically executes it for you.To put it in C++ terms, think of it as a "constructor" for your script.

Remember, loading a script involves a decent amount of hard drive access, format validation, and possibly even an entire compilation of the script (if your script is still in source code form). Scripts are no different than bitmaps or sounds in this respect; their loading phase is costly and should only be done outside of speed-critical code (i.e., outside of your main loop). Calling `lua_dofile ()` to execute a script on a per-frame basis would be frame rate homicide (which is only legal in Texas).

Getting back to the topic at hand, let's look at the script's constant declaration section:

```
ALIEN_COUNT          = 12;
MIN_VEL         = 2;
MAX_VEL         = 8;
ALIEN_WIDTH          = 128;
ALIEN_HEIGHT      = 128;
HALF_ALIEN_WIDTH   = ALIEN_WIDTH / 2;
HALF_ALIEN_HEIGHT   = ALIEN_HEIGHT / 2;
ALIEN_FRAME_COUNT   = 32;
ALIEN_MAX_FRAME   = ALIEN_FRAME_COUNT - 1;
ANIM_TIMER_INDEX   = 0;
MOVE_TIMER_INDEX   = 1;
```

The trick here is that Lua doesn't actually support constants. The best you can do is just pretend that it does by declaring your constant values as global variables that are written out with typical `CONSTANT_NOTATION` (like that). Lua just considers them typical globals, but at least your code will look the way you want it to. If you compare this block of code to the original hardcoded C version, you'll find that I've pretty much just copied the constant declarations and pasted them right into the Lua source.

Next up, let's have a look at your global variables

```
Aliens = {};
CurrAnimFrame = 0;
```

Only two declarations needed here. First you create a table called `Aliens` that will keep track of all of your bouncing heads. Next, you create a simple numeric called `CurrAnimFrame`, which keeps track of the current frame of the alien head animation.

With your constants and globals out of the way, you have all the data you need. Now it's time for some code. Let's have a look at the first of two functions this script will provide, `Init ()`:

```
function Init ()
       -- Initialize the alien sprites
       -- Loop through each alien in the table and initialize it
       for CurrAlienIndex = 1, ALIEN_COUNT do
              -- Create a new table to hold all of the alien's fields
              local CurrAlien = {};
              -- Set the X, Y location
              CurrAlien.X = GetRandomNumber ( 0, 639 - ALIEN_WIDTH );
              CurrAlien.Y = GetRandomNumber ( 0, 479 - ALIEN_HEIGHT );
              -- Set the X, Y velocity
              CurrAlien.XVel = GetRandomNumber ( MIN_VEL, MAX_VEL );
              CurrAlien.YVel = GetRandomNumber ( MIN_VEL, MAX_VEL );
              -- Set the spin direction
              CurrAlien.SpinDir = GetRandomNumber ( 0, 2 );
              -- Copy the reference to the new alien into the table
              Aliens [ CurrAlienIndex ] = CurrAlien;
       end
```

```
end
```

As you should remember, this is the function that's called by the following line back in the host application:

```
CallLuaFunc ( "Init", 0, 0 );
```

So, as soon as this line of code is hit, the `Init ()` function listed previously will be run.

The function really just has one job: initialize the array of bouncing alien heads. Just like in the original pure C version, this means giving each head a random location on-screen, a random velocity, and a random spin direction. Naturally, this is facilitated by a `for` loop.

To actually store the alien head demo, you need to store a smaller table at each index of the `Aliens` table. This is because there are a number of pieces of information that each head has to keep track of. To put this another way, think of it like a multidimensional array, or an array of `structs` in C. Each index of the table has another table (or rather, a *reference* to another table) that holds that particular element's information, like its X, Y location and its velocity. Check out Figure 6.13 for a visual representation of this.

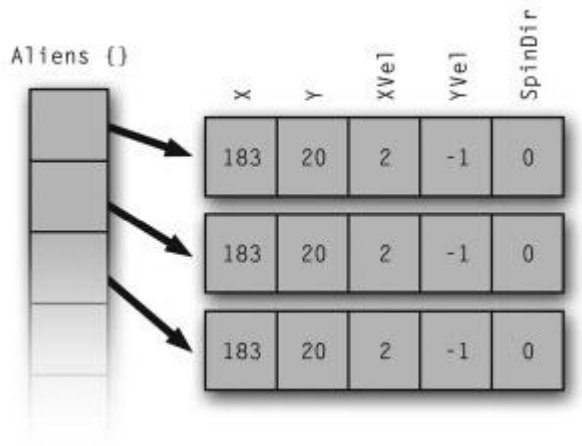

Figure 6.13: Each element of the `Aliens` table contains another table that holds that element's specific data.

All in all this is a simple concept, but there is one snag that can *really* trip you up if you're not ready for it. As I've mentioned before, it's important to think of tables in Luas references, rather than values. Because of this, assigning a table to an element of another table in a loop, like this:

```
Aliens [ CurrAlienIndex ] = CurrAlien;
```

means that `Aliens [ CurrAlienIndex ]` only receives a *reference* to the `CurrAlien` table, not the values themselves. So, at the next iteration of the loop, when you put new values into `CurrAlien` and assign it to the next index of `Aliens`, you'll find that both the current element as well as the previous element seem to suddenly have the same values. This is due to the fact that both elements have been given a reference to `CurrAlien`, so as soon as you change the values for the second element of the table in the next iteration of the loop, the first element will seem to inexplicably change along with it. Figure 6.14 illustrates this relationship.

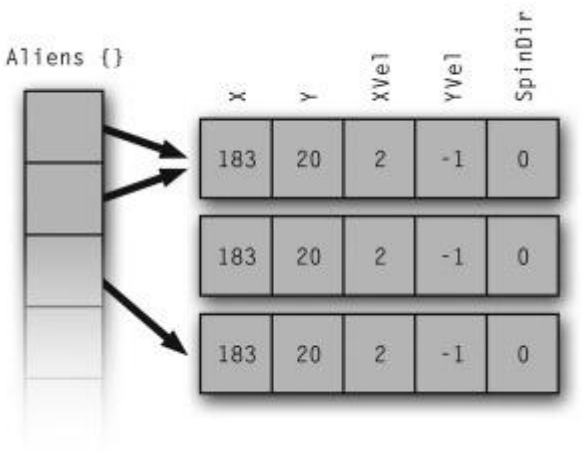

Figure 6.14: Two elements of `Aliens` point to the same table, and there fore reflect the changes made to one another.

To solve this problem, you simply start the loop with this line:

```
local CurrAlien = {};
```

Assigning `{}` to `CurrAlien` forces Lua to allocate a new table and therefore provide a fresh, unused reference. You can then fill the values of this instance of `CurrAlien` and freely assign it to the next element of `Aliens`, without worrying about overwriting the values you set in the last iteration. It's a simple problem with a simple solution, but left unchecked this little detail can cause logic errors that truly wreak havoc. :)

The rest of the alien head initialization loop is pretty much what you would expect; each element of `CurrAlien` is set to a random value, using the `GetRandomNumber ()` function that the previously discussed host API provides. Once this loop completes, `Init ()` is finished and the global `Aliens` table contains a record of every bouncing alien head.The script is now fully prepared to enter the main loop, which will call `HandleFrame ()` at each iteration. Let's have a look at this function:

```
function HandleFrame ()
	-- Blit the background image
	BlitBG ();
	-- Blit each sprite and move it along its path
	for CurrAlienIndex = 1, ALIEN_COUNT do
		-- Get the X, Y location
		local X = Aliens [ CurrAlienIndex ].X;
		local Y = Aliens [ CurrAlienIndex ].Y;
		-- Get the spin direction and determine
		-- the final frame for this sprite
		-- based on it.
		local SpinDir = Aliens [ CurrAlienIndex ].SpinDir;
		if SpinDir == 1 then
			FinalAnimFrame = ALIEN_MAX_FRAME - CurrAnimFrame;
		else
			FinalAnimFrame = CurrAnimFrame;
		end
		-- Blit the sprite
		BlitSprite ( FinalAnimFrame, X, Y );
	end
	-- Blit the completed frame to the screen
	BlitFrame ();
	-- Increment the current frame in the animation
	if GetTimerState ( ANIM_TIMER_INDEX ) == 1 then
		CurrAnimFrame = CurrAnimFrame + 1;
		if CurrAnimFrame >= ALIEN_FRAME_COUNT then
			CurrAnimFrame = 0;
		end
```

```
        end
        -- Move the sprites along their paths
        if GetTimerState ( MOVE_TIMER_INDEX ) == 1 then
               for CurrAlienIndex = 1, ALIEN_COUNT do
                      -- Get the X, Y location
                      local X = Aliens [ CurrAlienIndex ].X;
                      local Y = Aliens [ CurrAlienIndex ].Y;
                      -- Get the X, Y velocities
                      local XVel = Aliens [ CurrAlienIndex ].XVel;
                      local YVel = Aliens [ CurrAlienIndex ].YVel;
                      -- Increment the paths of the aliens
                      X = X + XVel;
                      Y = Y + YVel;
                      Aliens [ CurrAlienIndex ].X = X;
                      Aliens [ CurrAlienIndex ].Y = Y;
                      -- Check for wall collisions
                      if X > 640 - HALF_ALIEN_WIDTH or X <
                          -HALF_ALIEN_WIDTH then
                            XVel = -XVel;
                     end
                     if Y > 480 - HALF_ALIEN_WIDTH or Y <
                          -HALF_ALIEN_WIDTH then
                            YVel = -YVel;
                     end
                     Aliens [ CurrAlienIndex ].XVel = XVel;
                     Aliens [ CurrAlienIndex ].YVel = YVel;
              end
        end
end
```

Quite a bit larger than `Init ()`, eh? As you can see, there's a decent amount of logic to attend to here, so let's knock it out piece by piece.

The first step is easy; you make a single call to `BlitBG ()`, a host API function that slaps the background image into the framebuffer. This overwrites the last frame's contents and gives you a fresh slate on which to draw the new frame.

You then use a `for` loop to iterate through each alien in the bouncing alien head array, saving the X, Y location and final animation frame into local variables which are passed to host API function `BlitSprite ()` to put it on the screen. Notice that you don't necessarily use the global `CurrAnimFrame` as the frame passed to `BlitSprite ()`. This is because each head has its own spinning direction, which may be forwards or backwards. If it's forwards, you can use `CurrAnimFrame` as-is, but you must `subtract CurrAnimFrame` from `ALIEN_MAX_FRAME` if it's backwards. This lets certain sprites cycle through the animation in one direction, whereas others cycle through it the other way.

At this point, you've drawn the background image and each alien sprite. All that's left to complete this frame is to call `BlitFrame ()`, another host API function, which blasts the framebuffer to the screen. The graphical aspect of the current frame has been taken care of, but now you need to handle the logic. This means moving the alien heads along their paths and checking for collisions, among other things.

The first thing to do after blitting the new frame to the screen is update `CurrAnimFrame`. You do this by incrementing the variable, and resetting it to zero if the increment pushes it past `ALIEN_MAX_FRAME`. Of course, you want to perpetuate the animation at a fixed speed; if you incremented `CurrAnimFrame` every frame, the animation might move too quickly on faster systems. So, you've synchronized the speed of the animation with a timer that was created in the host application. This timer ticks at a certain speed, which means you have to use `GetTimerState ()` at each frame to see whether it's time to move the animation along. This ensures a more uniform speed across the board, regardless of frame rate.

This takes you to the last part of the `HandleFrame ()` function, which is the movement of each sprite and the collision check. Like the animation, the movement of the sprites is also synched to a timer, which means you make another call to `GetTimerState ()`. Assuming the timer has completed another tick, you start by saving the X, Y coordinates of the sprite and the X, Y velocities to local variables. You then add the velocities to the X, Y coordinates to find the next position along the path the alien should move to. You put these values back into the

`Aliens` array and then perform the collision check. If the new location of the sprite is above or below the extents of the screen, you reverse the Y velocity to simulate the bounce. The same goes for violations of the horizontal extents of the screen, which cause a reversal of the X velocity. Once these two checks have been performed, the X and Y velocities are placed back into the `Aliens` table as well and the movement of the sprites is complete.

You've now completed the script, which means the only thing left to do is sit back and watch it take off. Check out the demo on the accompanying CD. On the surface it looks identical to the hard-coded version, but there are two important differences. First, you may notice a slight speed difference. This is a valuable lesson—don't forget that despite all of its advantages, scripting is still noticably slower than native executable code in most situa tions. Second, and more obviously, remem ber that even though you've compiled the host application, the script itself can be updated and changed as much as you want without recompiling the executable.

> **Note** Remember, compiling your scripts with `luac` is always recommended. Now that you've finished working on the Lua demo, you might as well compile `script.lua` for future use.As I've said, `lua_dofile()` just needs the filename of the compiled version, and will handle the rest transparently. It costs you nothing, and in return you get faster script load times (although it's highly unlikely that you'll notice a difference in this particular example). Either way, it's a good habit to start early.

Because this is the whole reason you perhaps got into this crazy scripting business in the first place, I suggest you take the time to try changing the general behavior of the script and watch the executable change with it. As a challenge, try adding a gravity constant to the bouncing movement of the heads; perhaps something that will slowly cause them to fall to the ground. Once they're all at the bottom of the screen, reverse the polarity and watch them "fall" back up. This shouldn't take too much effort to implement given what you've done so far, and it will be a great way to experience first-hand the power scripts can have over their compiled host applications. Maybe you can create some trig functions in the host API and use them to move the gravity constant along a sinusoid.

Advanced Lua Topics

I've covered the core of the language as well as most of the details you'll need for integration. This should be more than sufficient for most of your game scripting needs, but if you're anything like me, you can't sleep at night until you've learned *everything*. And if you're anything like I am tonight, you won't sleep at all because you're all hopped up on *Red Bull* and are too busy running laps on the roof. So, allow me to discuss a few advanced topics that enhance Lua's power but are beyond the scope of this book:

- **Tag Methods**. One of Lua's defining features is the capability for it to extend itself. This is implemented partially through a feature called *tag methods*, which are functions defined by the script that are assigned to key points during execution of Lua code. Because these functions are called automatically by the Lua runtime, the programmer can use them to extend or alter the behavior of said code.
- **Complex Data Structures.** Lua only directly supports the table structure, but as you've seen, tables can not only contain any value, but can also contain references to other tables as well as functions. You can probably imagine how these capabilities lend themselves to the construction of higher-level data structures.
- **Object-Oriented Programming.** This is almost an extension of the last topic, but Lua is capable of implementing classes and objects through clever use of tables. Remember, tables can include function references, which gives them the capability to simulate constructors, destructors, and methods. Because functions can return table references as well, constructor functions can create tables to certain specifications automatically. Oh, the possibilities!
- **The Lua Standard Library.** Lua also comes with a useful standard library, much like the one that comes with C. This library is broken into APIs for string manipulation, I/O, math, and more. Becoming familiar with this library can greatly expand the power and flexibility of your scripts, so it's definitely worth looking into. Also, in case you were wondering, this is why your Lua distribution comes with `lualib.h` and `lualib.lib`. These extra files implement the standard library.

Web Links

For more general information on Lua, as well as the Lua user community, check out the following links. These are also great places to begin your investigation of the advanced topics described previously:

- **The Official Lua Web Site: `http://www.lua.org/`**. This is the official source for Lua documentation and distributions. Check here for updates on the language and system, as well as general news.

- **lua-users.org:** `http://www.lua-users.org/`. A gathering of a number of Lua users, offering a focused selection of content and resources.
- **lua-l: Lua Users Mailing List:** `http://groups.yahoo.com/group/lua-l/`. The lua-l Yahoo Group is a gathering of a number of Lua developers who discuss Lua news and ask/answer questions. It's a frequently evolving source of up-to-date Lua information and a good place to familiarize yourself with the language itself and its real-world applications.

Previous Next

Use of content on this site is expressly subject to the restrictions set forth in the Membership Agreement.
Books24x7 and Referenceware are registered trademarks of Books24x7, Inc.
Copyright © 1999-2005 Books24x7, Inc. - Feedback | Privacy Policy (updated 03/2005)