Arrays

En el capítulo anterior hemos visto en detalle las tablas, que son la estructura de datos básica de Lua. Las tablas son

extremadamente flexibles y permiten crear tipos específicos de estructuras de datos. En este nuevo capítulo vamos a ver con más detalle un tipo específico de tablas: los arrays.

Los arrays son estructuras de datos indexadas, formadas por una lista de pares de elementos, cada elemento tiene una clave numérica y un valor que puede ser de cualquier tipo, incluyendo otros arrays. Por lo tanto un array en Lua no es más que una tabla en la que las claves son numéricas.

Como las tablas, los arrays no tienen una longitud fija y pueden albergar tantos datos como los recursos disponibles lo permitan.

Arrays unidimensionales

> A diferencia de otros lenguajes, en Lua, los índices empiezan en 1 y no en 0. Por ello diremos que los arrays en Lua son de índice 1.

Los arrays unidimensionales representan una lista de un solo nivel. Se pueden crear en Lua definiendo una tabla simple:

```
ar = {"elemento 1", "elemento 2", "elemento 3"}


for i = 1, 3 do
  print(i, "-", ar[i])
end
```

En el ejemplo anterior, hemos definido una tabla con tres elementos de tipo cadena. Automáticamente, como no hemos definido una clave para cada uno de ellos, Lua le asigna una clave numérica incremental. Así *elemento 1* tendrá el índice 1 y *elemento 3* tendrá índice 3. Cuando ejecutos el código anterior tenemos la siguiente salida:

```
1       -       elemento 1
2       -       elemento 2
3       -       elemento 3
```

Si un índice de un array no existe, cuando intentamos accederlo, Lua devuelve `nil`:

```
> ar = {"elemento 1", "elemento 2", "elemento 3"}
> print (ar[4])
nil
```

Un array también puede ser definido como una tabla vacía e inicializarlo usando un bucle, por ejemplo:

```
ar = {}

for i = 1, 10 do
  ar[i] = i
end

for i = 1, 10 do
  print(ar[i])
end
```

El ejemplo anterior produce la siguiente salida:

```
1
2
3
4
5
6
7
8
9
10
>
```

Array multidimensional

Los arrays multidimensionales son aquellos que tienen otros arrays anidados. Un ejemplo de array multidimensional se muestra a continuación:

```
> md = {{"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}}
> print(md[1][2])
```

```
elemento 1.2
```

En la primera línea del ejemplo definimos un array de dos elementos cuyos valores son dos arrays de dos elementos cada uno. Para acceder al los elementos de cada array hay que poner grupos de corchetes según los niveles que queremos acceder. En nuestro ejemplo `md[1][2]`, el primer grupo de corchetes por la izquierda `[1]` representa al array padre, que contiene dos arrays; el segundo grupo `[2]`, representa a un elemento dentro del array hijo. Por ello `md[1][2]`, representa al elemento *elemento 1.2*, que corresponde con el segundo elemento del primer array.

Los arrays multidimensionales se pueden recorrer anidando los bucles también. Primeramente tendremos un bucle que recorerá el array padre y luego otro bucle anidado que recorrerá el array hijo. Observa el siguiente ejemplo:

```
 md = {{"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}}


for i = 1, 2 do
  for j = 1, 2 do
    print(i, j, md[i][j])
  end
end
```

Al ejecutar el código anterior se produce la siguiente salida:

```
1       1       elemento 1.1
1       2       elemento 1.2
2       1       elemento 2.1
2       2       elemento 2.2
```

Hasta ahora se ha presentado el caso de un array bidimensional, pero, no hay limitación en cuanto a los niveles de un array multidimensional.

Longitud de un array

Hasta ahora hemos mostrado en todos los ejemplos el valor constante de un array para recorrerlos con un bucle. En la mayoría de aplicaciones prácticas donde existan arrays de longitud variable, este método no es muy práctico.

El operador `#` de Lua permite recuperar la longitud de un array. Ahora, podemos reescribir uno de los ejemplos anteriores para que sea más dinámico y no tan rígido:

```
ar = {}
```

```
for i = 1, 10 do
  ar[i] = i
end

ar[#ar + 1] = #ar + 1

for i = 1, #ar do
  print(ar[i])
end
```

En el ejemplo anterior, inicializamos un array con 10 elementos del 1 al 10. En la siguiente línea usamos el operador `#` para introducir en la posición `# + 1` el valor `# + 1`, que corresponde con 11. De este modo podemos alterar de forma dinámica nuestro array, sin importar si tiene 5, 20 o 1000 elementos. Ahora cuando ejecutamos el código anterior obtenemos el siguiente resultado:

```
1
2
3
4
5
6
7
8
9
10
11
```

Asignación de valores en un array

Una vez inicializados los arrays, sus valores pueden ser modificados en todo momento. Observa los siguientes ejemplos:

```
ar = {"elemento 1", "elemento 2", {"elemento 3.1", {"elemento 3.2.1", "elemento 3.2.2"},
"elemento 3.3"}}

ar[2] = "modificado 2"
ar[3][2][1] = "modificado 3.2.1"
ar[3][3] = "modificado 3.3"

print(ar[2])
```

```
print(ar[3][2][1])
print(ar[3][3])
```

Al ejecutar el código anterior obtenemos:

```
modificado 2
modificado 3. 2. 1
modificado 3. 3
```

Revisión #1
Creado 16 octubre 2023 06:47:23 por Guillermo
Actualizado 16 octubre 2023 08:26:51 por Guillermo