

Funciones: haciendo tus propias herramientas

Este capítulo cubre:

- Escribir y usar funciones en ChuckK
- Definir y nombrar funciones
- Argumentos de función y tipos de retorno
- Funciones que se llaman a sí mismas
- Uso de funciones para composición de sonido y música

Ahora que te has divertido creando una máquina de ritmos usando el UGen `SndBuf`, es hora de aprender a escribir y usar funciones. Ya has usado un número de funciones, que también has llamado métodos, cuyo rango va entre cambiar `.freq` y `.gain` de osciladores, a usar `Std.mtof()` para convertir números de notas MIDI en frecuencias, a generar números aleatorios usando `Math.random2()` y `Math.random2f()`. Todos estos son ejemplos de funciones.

A menudo durante la escritura de programas necesitas hacer el mismo tipo de cosas múltiples veces y en múltiples lugares. Hasta el momento has reescrito o copiado bloques de código, posiblemente cambiando solo una pequeña parte. Has visto cómo el uso de las funciones de las bibliotecas `Standard` y `Math` incluidas en ChuckK te permiten hacer mucho trabajo llamándolas con un argumento o dos. ¿Que pasaría si pudiéramos crear y llamar a nuestras propias funciones? Eso es lo que aprenderás a hacer en este capítulo.

Añadir funciones a tus programas te permitirá dividir tareas comunes en

unidades individuales. Las funciones te permiten construir pequeños módulos de código que puedes usar una y otra vez. La modularidad ayuda a la reusabilidad y la colaboración con otros programadores y artistas. También hace que tu código sea de más fácil lectura.

En este capítulo aprenderás cómo escribir funciones, definirlas y nombrarlas, pasarles valores, y especificar su retorno. También veremos funciones que pueden llamarse a sí mismas, llamadas funciones recursivas. Estaremos atentos a funciones que hagan música y que mejoren aún más tus composiciones.

5.1 Creación y uso de funciones en tus programas

Inicialmente, estarás creando funciones en un solo archivo principal. Tendrás un archivo, grabado como `myProgram.ck`, por ejemplo, que contiene el programa principal que ejecuta tu código, y desde este código principal llamarás a tus funciones, cajas llamadas `Function 1` y `Function 2`, que son definidas de la misma forma que el archivo `myProgram.ck`. Como están definidas en el mismo archivo, puedes usarlas con su nombre cuando sean invocadas por tu programa y composición. Una vez que hayas definido y rprobado tus funciones, usualmente las moverás al final del archivo de tu programa, aunque puedes definir tus funciones en cualquier parte.

NOTA Es posible definir funciones y grabarlas en sus propios archivos, y estarás haciendo esto más adelante, pero por ahora mantendremos todo en el mismo archivo.

5.1.1 Declaración de funciones

Las funciones son como las variables, así que tienes que declararlas para poder usarlas, tal como lo hiciste con los `int`, `float` Y `SinOsc`, entre otros. Empezaremos discutiendo cómo declarar una función. Existen cuatro partes principales en la declaración de una función, como se muestra en la figura 5.2. Empiezas la línea con la palabra `function`, o con su abreviación `fun`. A continuación, declaras el tipo de retorno, o el tipo de datos que la función arroja tras su ejecución. Por ejemplo, esta función retornará un entero cuando termine su ejecución; esto es como `Std.ftoi()`, que retorna un entero a partir de un `float`.

NOTA Usar una función también recibe a veces el nombre de llamar o invocar una función.

A continuación, le darás un nombre a la función. Tal como en el caso de las variables, puedes nombrar las funciones como quieras, aunque es útil usar nombres con sentido y que impliquen la utilidad de la función. Para finalizar, haces una lista de los argumentos de entrada, los que son opcionales. Estas son variables que le pasas a la función para su uso durante la ejecución. Pueden haber muchos distintos argumentos de entrada distintos, todos de tipos distintos, basado en tus decisiones de diseño como programador.

5.1.2 Tu primera función musical

Observemos ahora una función útil musicalmente, una que retorna el intervalo entre dos números de notas MIDI

```
function int interval(int note1, int note2)
```

```
{  
    note1 - note2 => int result;  
    return result;  
}
```

Sí, esto es bastante trivial, porque podrías haber restado los dos números de notas, pero hacerlo en una función cumple con dos propósitos; puedes usarla una y otra vez y puedes nombrarla con algo que te recuerde por qué estás haciendo el cálculo, en este caso, interval.

Como puedes ver en la figura 5.3, puedes pensar en una función como un módulo de código con valores de entradas y un valor de salida. Este tiene dos entradas de variables enteras llamadas note1 y note2. La función usa esos valores y crea un resultado de valor entero que es luego entregado al programa que llamó a la función. Por ejemplo, si pasas (72, 60) como entrada, como los argumentos, la función interval retorna 12 (72-60), el número de pasos en una octava.

Para entender mejor este proceso, empecemos con un ejemplo aún más simple, una función que sube en una octava cualquier número de nota MIDI que se le pasa, como se muestra en el listado 5.1. Primero creas una función llamada addOctave (1) con un argumento de entrada entero que llamas note y declaras un tipo de retorno int. Luego creas un resultado entero llamado result (2), que puedes usar para almacenar tu respuesta final. Sumas 12 (una octava) a tu variable note (3) y almacenas ese nuevo valor en tu variable result. La variable result es retornada al programa principal.

Cuando el programa empieza, siempre parte en el programa principal y no corre la función hasta que es llamada. El programa empieza donde addOctave() es llamada (5). Observa que addOctave() tiene un par de

paréntesis, y aquí puedes ingresar cualquier entrada que quieras (pero debe ser un entero). En este caso empiezas pasando el número 60, y la función es ejecutada usando 60 como argumento y corriendo la función. Al final de la función, el resultado es retornado, enviando 72 a answer (6).

Listado 5.2 Definición y prueba de una función que suma una octava a cualquier número de nota MIDI

```
//Una función simple de ejemplo

//declaramos nuestra función aquí
//(1) Declaración de la función
fun int addOctave(int note)
{
    //(2) Resultado a retornar
    int result;
    //(3) Calcula el valor a retornar
    note + 12 => result;
    //(4) Lo retorna
    return result
}

//Programa principal de prueba de addOctave, llama e imprime e
//(5) Usa la función
addOctave(60) => int answer;

//(6) Revisa el resultado
<<< answer >>>;
```

answer imprime esto en la consola:

```
72 :(int)
```

Llamar a `addOctave` con un argumento de 72 retorna 84, 90 retorna 102, y así. Definir una función como `addOctave` te permite usarla una y otra vez y como tiene un nombre lleno de significado, puedes saber qué estás haciendo cada vez que la usas o la vez en el código.

Dos maneras de llamar funciones en ChuckK

Las funciones con un argumento pueden ser llamadas de dos maneras:

```
addOctave(60);
```

o

```
60 => addOctave;
```

Estas dos maneras de invocar una función funcionan exactamente de la misma manera.

Hagamos una prueba simple sonora/musical de la función `addOctave` y probemos también dos maneras de llamar funciones, añadiendo estas líneas al listado 5.1. En el listado 5.2 tocas un tono sinusoidal (1) en el C central (nota MIDI 60, asignada a una variable entera llamada `note` (2)). Primero usa la forma con paréntesis de llamar a `Std.mtof` (3) y luego usa la forma "hacer ChuckKing" de la función `addOctave` y la función `Std.mtof` (4). Como puedes ver, las dos funcionan bien.

Listado 5.2 Prueba con sonido de la función `addOctave`

```
//Usemos la función addOctave para hacer música
//(1) Oscilador para poder escuchar la función addOctave
SinOsc s => dac;
//(2) Nota inicial
60 => int myNote;

//(3) Toca la nota inicial
Std.mtof(myNote) => s.freq;
second => now;

//(4) Toca una octava más arriba
myNote => addOctave => Std.mtof => s.freq;
second => now;
```

Ahora volvamos atrás y probemos la función interval. Pasa dos argumentos de entrada a la función, note1 y note2, como se muestra en el siguiente listado. La función es ejecutada dps veces, con dos pares de números distintos, que luego calculan result, dos veces, retornándolo al programa principal para ser impresos, resultando esto en la consola:

```
12 -7
```

Listado 5.3 Definición y prueba de una función de intervalo MIDI

```
//Definición de la función
fun int interval(int note1, int note2)
{
    note2 - note1 => int result;
    return result;
}

//programa principal, prueba e imprime
```

```
interval (60, 72) => int int1;  
interval (67, 60) => int int2;  
  
<<< int1, int2 >>>
```

5.1.3 Variable globales versus locales

En la versión modificada del programa mostrada en el listado 5.4, es importante entender que las variables globales pueden ser usadas en cualquier lugar, incluyendo dentro de las funciones o estructuras con un par de llaves {}; las variables locales de la función, en este caso, `result`, no pueden ser llamadas fuera del ámbito (scope).

Cada variable tiene un ámbito basado en dónde fue definida, llamado localidad. En cada función, existe un par de llaves {}. Esto define el área del ámbito local. Entonces en los programas 5.4 y 5.5, la función `interval` posee las variables de ámbito local `note1`, `note2` y `result`. El programa principal, como se muestra en el listado 5.4, posee las variables de ámbito global `int1`, `int2`, `glob` y `howdy`.

El programa en el siguiente listado dará un error, "line 16: undefined variable `result`" (línea 16: resultado de variable indefinida). Pero si decides borrar la última línea, o comentarla al insertar `//` al principio, el programa correrá sin problemas.

Listado 5.4 Ámbito de variables locales versus globales

```
//define algunas variables globales  
"HODY" => string howdy;  
100.0 => float glob;  
int int1, int2;
```



```
//Definición de la función
fun int interval(int note1, int note2)
{
    int result;
    note2 - note1 => result;
    <<< howdy, glob >>>;
    return result;
}

//Programa principal, prueba e imprime
interval(60, 72) => int1;
interval(67, 60) => int2;

<<< int1, int2 >>>;

<<< result >>>; //Esta línea causará el error
```

Hasta el momento has hecho y usado muchas funciones simples que operan en enteros interpretados como números de notas MIDI. Ahora harás y usarás algunas nuevas funciones tipo float para ganancia y frecuencia.

5.2 Algunas funciones para calcular ganancia y frecuencia

Ahora que has visto los fundamentos de las funciones y los has usado para resolver un par de problemas simples, pongámoslos en acción para controlar sonido. Definirás funciones que operan en floats, interpretados como ganancias (0.0 a 1.0) y frecuencias, y luego los usarás para hacer que tus programas sean más expresivos musicalmente. Configurarás

ganancias y frecuencias de oscilador usando tus nuevas funciones. Luego verás cómo definir y usar una función que aumente y disminuye gradualmente como una rampa, creando una envolvente de amplitud suave para cada nota que tocas. Luego volverás a usar un SndBuf para reproducir un archivo de sonido, pero con la adición de una función que hace cortes aleatorios en el archivo, granulariza, mientras se reproduce.

Comencemos con un programa simple, mostrado en el listado 5.5, que tiene una función llamada halfGain (mitad de la ganancia) (2), que toma un valor de entrada de punto flotante llamado originalGain y arroja una salida de punto flotante. Como puedes ver, esta función es extremadamente simple, solo divide la entrada por la mitad antes de ser retornada al programa principal. Para usarla, haz un SinOsc s conectado al dac (1). Después salta al programa principal; la ejecución omite la definición de la función hasta que la función es llamada de forma explícita. Luego imprimies el valor actual de s.gain() (3). Observa que cuando llamas a s.gain() con un con junto vacío de paréntesis, un método interno de SinOsc retorna el valor actual de la ganancia .gain de s. Espera por un segundo, dejando que la onda sinusoidal toque a su volumen inicial. Luego llama a la función p con la entrada s.gain() (4). La función se ejecuta, diviendo originalGain por la mitad y retornando el nuevo valor de 0.5 para definir s.gain. De nuevo espera 1 segundo mientras el objeto s es reproducido al nuevo volumen más bajo.

Listado 5.5 Función para dividir la ganancia (o cualquier float) por la mitad

```
//(1) Oscilador para probar la función halfGain
SinOsc s => dac;

//nuestra función
```

```
//(2) Define la función halfGain
fun float halfGain(float originalGain)
{
    return (originalGain * 0.5);
}

//recuerda que .gain es una función interna de SinOsc
//(3) Imprime la ganancia inicial de SinOsc
<<< "ganancia completa: ", s.gain() >>>;

second => now;

//llamada a halfGain()
halfGain(s.gain()) => s.gain;
//(4) Imprime la nueva ganancia de SinOsc tras ser dividida po
<<< "la ganancia es ahora la mitad: ", s.gain() >>>;
second => now;
```

5.2.1 Hacer música real con funciones

A continuación, vas a construir un ejemplo musical real que usa tres diferentes osciladores de onda cuadrada: s, t y u. Esta vez quieres usar las funciones para ayudar a configurar las frecuencias de todos los osciladores. Primero, define dos funciones, `octave()` y `fifth()` (octava y quinta):

```
//Funciones para octava y quinta
fun float octave(float originalFreq)
{
    return 2.0 * originalFreq;
}
```

```
fun float fifth(float originalFreq)
{
    return 1.5 * originalFreq;
}
```

Observa que estas funciones tienen el mismo nombre, `originalFreq`, para el argumento de entrada. Como el ámbito de ambas variables es solo local a cada una de las funciones, ¡todo está bien! El `originalFreq` en la función `octave()` solo puede ser visto dentro de su propio ámbito local, tal como el `originalFreq` de la función `fifth()` que solo puede ser visto localmente a esa función. Para evitar confusión, probablemente no quieres nombrar alguna variable local con el nombre `originalFreq`.

Si escarbas un poco más hondo, verás que la nueva función `octave()` es diferente de la anterior (que aceptaba un número de nota MIDI entero). Esta toma la variable de entrada y la multiplica por 2.0. La función espera un valor de frecuencia en Hertz. La teoría acústica postula que un salto de una octava ocurre cada vez que se duplica la frecuencia. La función `fifth()` multiplica por 1.5 resultando en un intervalo musical llamado "quinta justa" (el nombre se debe a que es la quinta nota en una escala musical standard) sobre cualquier argumento de frecuencia.

Revisemos ahora un programa completo que usa ambas funciones, para crear un barrido ascendente en frecuencia de riqueza sonora, como se muestra en el listado 5.6. Primero haces tres osciladores y los conectas a los canales izquierdo, central y derecho del dac (1). Luego defines la ganancia de todos los osciladores (2), para que cuando sean sumados, en total no excedan 1.0, lo podría casuar que el dac o los parlantes se saturen y suenen mal. Observa que te aprovechas de una característica de ChuckK que permite que los tres osciladores sean configurados al mismo valor (2);

definir un parámetro o valor de variable también retorna el mismo valor. El programa principal gira en torno a un bucle for que aumenta de 100 a 500 en incrementos de 0.5 (3). Cada vez que ocurre, el valor de la variable freq del bucle for es usada para definir la frecuencia del oscilador SqrOsc s (4). Luego usas el valor de retorno de la función octave() para definir la frecuencia de t (6), y usas la función fifth() para definir la frecuencia de u (6).

Listado 5.6 Uso de funciones para definir frecuencias de osciladores

```
//tres osciladores en stereo
//(1) Tres ondas cuadradas, paneadas a izquierda, centro y derecha
SqrOsc s => dac.left;
SqrOsc t => dac.;
SqrOsc u => dac.right;

//define las ganancias para no sobrecargar ael dac
//(2) Define las tres ganancias
0.4 => s.gain => t.gain => u.gain;

//funciones octave y fifth
fun float octave(float originalFreq)
{
    return 2.0 * originalFreq;
}

fun float fifth(float originalFreq)
{
    return 1.5 * originalFreq;
}

//programa principal
//(3) Barrido en frecuencia de 100 a 500 en pasos de 1/2 Hz
for (100 => float freq; freq < 500; 0.5 +=> freq)
{
```

```
//(4) Define la frecuencia de la onda cuadrada izquierda como  
freq => s.freq;  
//(5) Define la frecuencia de la onda cuadrada central una octava  
octave(freq) => t.freq;  
//(6) Define la frecuencia de la onda cuadrada derecha una quinta  
fifth(freq) => u.freq;  
<<< s.freq(), t.freq(), u.freq() >>>;  
10 :: ms => now;  
}
```

Has visto dos ejemplos de cómo las funciones pueden ser usadas para ayudar a manipular métodos `.gain` y `.freq` de tus unidades generadores tipo oscilador. No obstante, has estado controlando `.freq` y `.gain` sin usar tus propias funciones, ¿no es cierto? Es agradable, por otro lado, haber nombrado con sentido las funciones, incluso si hacían algo que pudiste haber hecho de otra manera, porque cualquiera que lea tu código (incluyéndote a ti en el futuro) puede adivinar qué están haciendo las funciones `octave` y `fifth`.

5.2.2 Uso de una función para cambiar gradualmente parámetros sonoros

En el siguiente ejemplo te mostraremos cómo hacer algo que sería mucho más difícil y menos flexible si no usaras una función. Vamos a definir una función para cambiar gradualmente el volumen de un oscilador; haciendo una rampa hacia arriba y abajo a una tasa arbitraria. Si revisas el programa del listado 5.7 verás nuestra nueva función `swell()` (1). Observa que tiene cuatro argumentos: un `UGen` llamado `osc` y tres entradas de punto flotante: `begin`, `end` y `step`. Las últimas tres variables son usadas para controlar dos bucles, uno para aumentar (2) y otro para disminuir (3) el

volumen del oscilador osc.

NOTA El tipo de retorno de la función swell es void, lo que significa que no retorna valor alguno, porque no lo necesita. ¿Recuerdas el capítulo 1 cuando introducimos tipos de datos, y te prometimos que usaremos void después? Bueno, aquí estamos. También puedes hacer funciones con argumentos void, como lo hiciste con s.gain() en el listado 5.6, porque no necesita una entrada de datos para cumplir su labor. Algunas funciones como esta tienen un valor de retorno, otras realizan tareas útiles sin pedir ni arrojar datos. Esto podría servir para variables globales, entre otros.

Listado 5.7 Uso de una función swell para hacer rampas arriba y abajo del volumen de un oscilador

```
//función swell, opera sobre cualquier tipo de UGen
//(1) definición de la función swell
fun void swell(UGen osc, float begin, float end, float step)
{
  float val;
  //aumenta el volumen
  //(2) bucle for para hacer una rampa ascendente de volumen
  for (begin => val; val < end; step+=val)
  {
    val => osc.gain;
    0.01 :: second => now;
  }

  //disminuye el volumen
  //(3) bucle for para hacer una rampa descendente de volumen
  while (val < begin)
  {
    val => osc.gain;
    step -=> val;
  }
}
```

```
    0.01 :: second => now;  
  }  
}
```

NOTA Cuando definimos nuestra función `swell()`, especificamos que el primer argumento sea un `UGen`, lo que significa que le puedes pasar a la función absolutamente cualquier unidad generadora en ese lugar. Esto se aprovecha de una propiedad llamada herencia, sobre la que aprenderemos más en un capítulo más adelante. Por ahora, puedes usarla para hacer que tus funciones sean extremadamente flexibles y mucho más reusables. También aprenderás sobre muchos más tipos de `UGens`, a partir del próximo capítulo.

Si revisas nuestro programa principal, mostrado en el listado 5.8, que usa `swell`, verás una cadena de audio oscilador `=> dac` (1) y un arreglo que usaremos para tocar una melodía (2). Luego entras a un bucle para tocar todas las notas del arreglo (3). En cada oportunidad, se llama a la función `swell`(5). Observa que `swell()` hace que el tiempo transcurra dentro de ella. Es importante entender que el programa principal salta a la función en (4) y procede a ejecutar cada línea de código. El tiempo pasa en esta función mientras el volumen cambia, y cuando la función termina, la función retorna al programa principal, que ejecuta otra vez el bucle.

Listado 5.8 Programa principal que usa `swell` de forma expresiva para tocar una melodía

```
//programa principal  
//nuestro patch sonro  
//(1) oscilador de onda triangular para probar la función swell  
TriOsc tri => dac;
```



```
//(2) arreglo global de las notas a tocar
[60, 62, 63, 65, 63, 64, 65, 58, 57, 56] @=> int notes[];

//función swell aplicada a cada nota
//(3) itera sobre las notas del arreglo
for (0 => int i; i < notes.cap(); i++) {
    //(4) define la frecuencia
    Std.mtof(notes[i]) => tri.freq;
    //(5) llamada a la función swell
    swell(tri, 0.2, 1.0, 0.01);
}
```

Como puedes ver, esta es una función altamente expresiva que puede ser usada para convertir un oscilador simple en un instrumento musical con comienzos y finales suaves de notas individuales.

5.2.3 Granularizar: una función licuadora de audio para SndBuf

En un ejemplo final para esta sección, usarás conceptos que has aprendido en el capítulo 4 (sobre samples y archivos de audio), pero ahora usando una función. El programa en el listado 5.9 carga y reproduce un SndBuf (1) pero constantemente lo secciona, reproduciendo pedazos aleatorios usando el método .pos() (5). Esto es una forma de síntesis y manipulación sonora llamada síntesis granular, que ha existido hace un tiempo; incluso antes de la era digital, la gente cortaba y pegaba pedazos de cinta de audio para realizar síntesis granular.

En el listado 5.9, primero haces un SndBuf2 llamado click, lo conectas al dac (1), y cargas un archivo de sonido stereo (2). Recuerda el el director de

audio que contiene tus archivos de audio debe estar en el mismo lugar que el programa. Para aprovechar las funciones, crea una nueva llamada `granularize()` (3), que toma un argumento llamado `myWav` (cualquier `SndBuf`) y un entero llamado `steps`. Esta función usa `steps` para crear granos aleatorios (pequeñas secciones) de sonido a partir de `myWav`. Para lograrlo, toma el número total de samples en el archivo de sonido y lo divide por la variable `steps` para obtener un tamaño de grano (4). Luego puedes seleccionar una posición aleatoria de reproducción dentro del archivo de sonido (5). Luego avanzas el tiempo según `grain` y retornas al bucle principal, que será ejecutado otra vez y para siempre.

Listado 5.9 Crear y usar una función `granularize()` para cortar un archivo de sonido

```
//(1) crea un SndBuf2 stereo y lo conecta
SndBuf2 click => dac;

//(2) carga un archivo stereo de sonido
me.dir*() + "/audio/stereo_fx_01.wav" => click.read;

//función para seccionar cualquier archivo de sonido
//(3) define la función granularize
function void granularize(SndBuf myWav, int steps)
{
  //(4) calcula el tamaño del grano
  myWav.samples() / steps => int grain;
  //(5) Hace que el puntero apunte a una posición aleatoria de
  Math.random2(0, myWav.samples() - grain) + grain => myWav.pos;
  grain :: samp => now;
}

//programa principal
while (true)
```

```
{  
    //llama a la función, aquí transcurre el tiempo  
    granularize(click, 70);  
}
```

Además de los asombros sonidos que hace tu función `granularize()`, lo maravilloso es que esta función puede tomar cualquier archivo de sonido y seccionarlo, para luego ser reusado una y otra vez. Todo lo que tienes que hacer es cargar un archivo de sonido distinto.

PRUEBA ESTO Carga distintos archivos de sonido en el objeto `click` `SndBuf2` del listado 5.9. Hay otros archivos stereo y de mayor duración en el directorio de audio que has estado usando, pruébalos. Cambia el argumento `steps` y escucha la diferencia. Además, encuentra otros archivos de sonido (.wav, .aiff) en tu computador, cópialos al directorio de audio, cárgalos en `click`, y escucha cómo suenan cuando son granularizados.

5.3 Funciones para construir formas de composición

Has empezado a ver cómo puedes usar funciones para brindar un gran control expresivo y de maneras que puedes volver usar. En esta sección verás cómo las funciones pueden ser usadas para ayudar a crear nuevas formas de composición. Primero crearás un patrón melódico, luego aprenderás cómo puedes usar funciones para operar en arreglos, tanto leyendo como modificando los elementos internos. Finalmente, construirás una máquina de ritmos súper flexible usando funciones y arreglos.

5.3.1 Tocar una escala con funciones y variables globales

Exploremos cómo las funciones y las variables globales pueden trabajar en conjunto, escribiendo un programa que camina hacia arriba y abajo a través de un pequeño patrón de escala, subiendo siempre en altura. En el listado 5.10, usarás un nuevo UGen generador de sonido, llamado Mandolin (mandolina), y lo conectarás al dac (1). Mandolin es un "instrumento" completo que puedes tocar con comandos simples como .freq (al igual que SndBuf) y .noteOn (que esencialmente toca una nota en la Mandolin). Hablaremos más sobre Mandolin y otros UGens a partir del próximo capítulo. Continuando en el listado 5.10, también defines una variable global llamada note y la inicializas a 60 (C central) (2). Tanto mand como note son globales, porque las declaras al principio de tu programa, afuera de cualquier par de llaves. A continuación, defines dos funciones: noteUp() (3) y noteDown (7), que no tienen argumentos de entrada, porque solo operan sobre variables globales. También observa que los tipos de salida son void (lo que denota que no tienen un valor de retorno). Si ahora observas con mayor detención a noteUp() (3), verás que actualiza la variable global note, sumándole uno (4), e imprime el nuevo valor de note (5). noteDown (7) tiene una funcionalidad similar, pero resta 1 a note (8). Defines una función adicional llamada play() (9), que define la altura de tu mand (10), la toca con noteOn (11), y avanza el tiempo en 1 segundo (12). Las dos funciones noteUp() y noteDown() llaman (6) a la función play() (9) para hacer que la mandolina toque.

Listado 5.10 Funciones void sobre variables globales, para diversión escalar musical

```

//variables globales
//(1) crea y conecta un UGen instrumento tipo Mandolin
Mandolin mand => dac;
//(2) variable global note para nota
60 => int note;

//funciones
//(1) definición de la función noteUp
fun void noteUp()
{
    //(4) suma 1 a la variable global note
    1 +=> note;
    //(5) la imprime
    <<< note >>>;
    //(6) la reproduce
    play();
}

//(7) definición de la función noteDown
fun void noteDown()
{
    //(8) resta 1 a la variable global note
    1 -=> note;
    <<< note >>>;
    play();
}

//(9) define la función play
fun void play()
{
    //(9) define la frecuencia de la Mandolin global usando note
    Std.mtof(note) => mand.freq;
    //(10) Toca la nota en la Mandolin
    1 => mand.noteOn;
    //(11) Hace transcurrir one segundo antes de volver al bucle
    second => now;
}

```

```
}
```

Observemos ahora el listado 5.11, que es el bucle infinito principal del programa (1) que usa nuestras funciones `noteUp()` y `noteDown()`. Observa que el bucle no hace que el tiempo transcurra. En muchos casos, esto significa que el programa no funcionará y que Chuck se quedará colgado. Pero como viste en los ejemplos anteriores de `swell()` y `granularize()`, el tiempo puede avanzar dentro de las funciones. En este caso, recuerda que cuando llamamos a `noteUp()` (2), luego de que realiza su tarea, llama a la función `play()` ((9) en el listado anterior), que realiza su trabajo y hace que el tiempo avance 1 segundo. Después de que esto termina, `play` retorna a `noteUp()`, que inmediatamente retorna al programa principal, donde se llama a `noteDown()` (3). `noteDown()` hace que `note` decrezca, la imprime, llama a `play()` y luego retorna. El proceso continua en las siguientes llamadas a `noteUp()` y `noteDown()`.

Listado 5.11 Uso de las funciones `noteUp` y `noteDown` dentro de un bucle principal

```
//programa principal, "melodía" ascendente gradualmente
//(1) programa principal para probar las funciones noteUp y noteDown
while (true) {
    //(2) llamada a noteUp
    noteUp();
    //(3) llamada a noteDown
    noteDown();
    //dos llamadas a noteUp
    noteUp();
    noteUp();
    //una llamada a noteDown y se repite el bucle
    noteDown();
}
```

```
}
```

Este proceso se sigue repitiendo y este programa toca las notas e imprime lo mostrado a continuación:

```
61 :(int)
60 :(int)
61 :(int)
62 :(int)
61 :(int)
62 :(int)
61 :(int)
62 :(int)
63 :(int)
62 :(int)
63 :(int)
... etc ...
```

5.3.2 Cambiar alturas de escalas usando una función sobre un arreglo

Ahora aprenderás cómo puedes usar arreglos con funciones. Tal como pasaste UGens como argumentos en funciones, los arreglos también son argumentos válidos. Por ejemplo, puedes definir una función llamada `arrayAdder` que modifica un miembro (`index`, por índice) de un arreglo (`temp`), para que tenga un nuevo valor (súmale uno). Usarás esto pronto para cambiar las alturas de un arreglo de notas llamado `scale`.

```
fun void arrayAdder(int temp[], int index) {
  1 +=> temp[index];
```

```
}
```

El siguiente listado prueba esta nueva función, declarando un arreglo global (1), y nuestra función `arrayAdder()` y luego probándola unas cuantas veces, modificando dos elementos del arreglo.

Listado 5.12

```
//(1) arreglo global de notas
[60, 62, 63, 65, 67, 69, 70, 72] @=> int scale[];

//función arrayAdder para modificarllo
fun void arrayAdder(int temp[], int index)
{
    1 +=> temp[index];
}

//hacer pruebas
<<< scale[0], scale[1], scale[2], scale[3] >>>;
arrayAdder(scale, 2);
<<< scale[0], scale[1], scale[2], scale[3] >>>;
<<< "scale[6] = ", scale[6] >>>;
arrayAdder(scale, 6);
<<< "scale[6] = ", scale[6] >>>;
```

La consola imprime

```
60 62 63 65
60 62 64 65
scale[6] = 70
scale[6] = 71
```


mostrando que los elementos del arreglo global han sido realmente modificados; array[2] cambió de 63 a 64, y array[2] de 70 a 71.

Más sobre ámbito (scope): copias temporales de argumentos int y float dentro de arreglos

Cuando passa un int o float a un arreglo, la variable dentro de la función es una copia del valor pasado, local solo a esa función. Este programa:

```
//variable global entera
60 => int glob;
//función que suma uno al argumento
fun void addOne(int loc)
{
    1 +=> loc;
    <<< "copia local de loc = ", loc >>>;
}
//
addOne(glob);
//
<<< "versión global de glob = ", glob >>>;
```

Imprime en la consola:

```
copia local de loc = 61
versión global de glob = 60
```

Esto claramente demuestra que aunque la variable local loc es modificada, la variable global pasada como argumento no es modificada. Para los adeptos a la ciencia por computación, esto se

llama pasar por valor, donde el valor de glob es copiado a una variable loc localmente declarada. Los arreglos son diferentes porque son pasados a las funciones por referencia, y el nombre variable de arreglo local es solo una referencia al exactamente mismo arreglo que fue pasado.

En otras palabras, cuando pasas la mayoría de los tipos de datos a una función, necesitas usar explícitamente la palabra clave return para obtener un resultado. Pero cuando pasas un arreglo a una función, los contenidos del arreglo son modificados de forma directa.

Ahora usemos nuestra función `arrayAdder()` para un propósito musical. El listado 5.13 crea una Mandolin (1) con la que toca una escala (2), luego usa la función `arrayAdder()` (3) para convertir el arreglo de la escala original a una escala distinta. Usando una nueva función `playScale()` (4) que creaste tocas las notas de cualquier arreglo de enteros pasado como un argumento, tocando la escala como fue originalmente creada (5), luego mueve los elementos segundo y sexto uno hacia arriba (6), y luego vuelve a tocar la escala (7). Observa que llamas a la función `arrayAdder()` dos veces (6) y puedes llamarla las veces que quieras y con distintos argumentos. Esto es el poder de las funciones, pueden ser usadas una y otra vez.

NOTA Para los aficionados a la teoría musical, convertimos una escala menor en modo dorio a una escala mayor standard.

Listado 5.1 Uso de la función `arrayAdder()` para convertir una escala menor a una mayor

```
//crea una mandolina y la conecta a la salida de audio
//(1) instrumento Mandolin
Mandolin mand => dac;
```

```

//arreglo global scale
//(2) arreglo scale de notas de números de nota
[60, 62, 63, 65, 67, 69, 70, 72] @=> int scale[];

//(3) definición de la función arrayAdder
fun void arrayAdder(int temp[], int index)
{
    1 +=> temp[index];
}

//(4) definición de la función playScale
fun void playScale(int temp[])
{
    for (0 => int i; i < temp.cap(); i++)
    {
        Std.mtof(temp[i]) => mand.freq;
        <<< i, temp[i] >>>;
        1 => mand.noteOn;
        0.4 :: second => now;
    }
    second => now;
}

//toca nuestra escala en nuestra mandolina
//(5) pone a prueba playScale
<<< "escala original" >>>;
playScale(scale);

//modifica nuestra escala
//(6) llama a arrayAdder para modificar dos elementos
arrayAdder(scale, 2);
arrayAdder(scale, 6);

//(7) cuando llamamos nuevamente a playScale, suena diferente
<<< "escala modificada" >;

```

```
playScale(scale);
```

5.3.3 Construir una máquina de ritmos con funciones y arreglos

Ahora que sabes cómo usar tanto variables globales como arreglos con funciones, empecemos a revisar cómo hacer un programa que le da forma a tus composiciones. En el listado 5.14, empiezas definiendo buffers de sonidos bombo (bass drum) y caja (snare drum) (1). A continuación lees los archivos de sonido (2) y defines a sus punteros al final (2) (para que no emitan sonido al principio). Luego define arreglos que usarás para controlar las secuencias de reproducción de tus samples de batería (4). En la función `playSection`(5), donde sea que pongas un 1 en estos arreglos, escucharás un sonido, y un 0 corresponderá a silencio. El bucle principal (6) llama a `playSection` con distintos arreglos de patrones (7) para hacer un bucle de batería.

Listado 5.14

```
//cadena de sonido: dos tambores
//(1) SndBufs para sonidos de bombo y caja
SndBuf kick => dac;
SndBuf snare => dac;

//carga los archivos de sonido para nuestros tambores
//(2) carga los archivos wav de bombo y caja
me.dir() + "/audio/kick_01.wav" => kick.read;
me.dir() + "/audio/snare_03.wav" => snare.read;

//(3) define sus posiciones iniciales a 0, para que no emitan
```

```

kick.samples() => kick.pos;
snare.samples() => snare.pos;

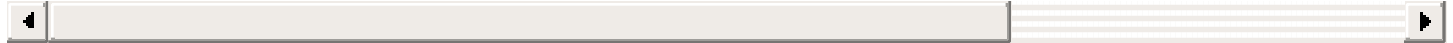
//(4) arreglos para almacenar valores lógicos, tocar = 1, no t
[1, 0, 0, 0, 1, 0, 0, 0] @=> int kickPattern1[];
[0, 0, 1, 0, 0, 0, 1, 0] @=> int kickPattern2[];
[1, 0, 1, 0, 1, 0, 1, 0] @=> int snarePattern1[];
[1, 1, 1, 1, 0, 1, 1, 1] @=> int snarePattern2[];

//(5) define la función playSection, argumentos tipo arreglo p
fun void playSection(int kickA[], int snareA[], float beattime
{
    for (0 => int i; i < kickA.cap(); i++)
    {
        if (kickA[i])
        {
            0 => kick.pos
        }
        if (snareA[i])
        {
            0 =<> snare.pos;
        }

        beattime :: second => now;
    }
}

//(6) bucle infinito de prueba
while (true)
{
    //(7)llama a playSection con diferentes patrones
    playSection(kickPattern1, snarePattern2, 0.2);
    playSection(kickPattern2, snarePattern2, 0.2);
    playSection(kickPattern1, snarePattern2, 0.2);
    playSection(kickPattern2, snarePattern1, 0.2);
}

```



EJERCICIO ¡Personaliza tu máquina de ritmos! Modifica los 1s y 0s de los arreglos, cambia los contenidos de los arreglos usados en las llamadas a `playSection()`, y cambia el tempo por medio la modificación del último argumento en las llamadas a `playSection()`. Añade más tambores - ¡eres el compositor y el programador!

Hasta el momento en este capítulo, usando funciones y arreglos, has aumentado tu expresividad y también haz tu código más flexible y de más fácil lectura. Pero ahora puedes ir más allá, por supuesto.

5.4 Recursión (funciones que se llaman a sí mismas)

Has aprendido los fundamentos de las funciones y cómo pueden transformar tu música y la arquitectura de tus programas. Ahora es tiempo de aprender técnicas avanzadas que pueden resultar en materiales sonoros y estructurales muy interesantes. Has visto cómo las funciones pueden llamar a otras funciones (como cuando `noteUp` y `noteDown()` llamaron a `play()` en el listado 5.10). ¿Pero pueden las funciones llamarse a sí mismas? ¿Y por qué habrías de querer hacerlo? Existen muchas estructuras musicales (como las escalas) y figuras (como trinos y redobles de tambores) que son realmente eventos o transformaciones repetidos. Muchas de estos podrían beneficiarse de una función que se pueda llamar a sí misma.

Aquí introducimos el concepto de recursión, que en programación significa que cosas útiles pueden ocurrir cuando una función se puede llamar a sí misma. Como las funciones te permiten realizar mucho trabajo, a veces de

forma repetida, entonces una función llamándose a sí misma podría multiplicar su poder dramáticamente. El clásico ejemplo de recursión enseñado en casi cada libro de programación es la función matemática factorial, pero como somos artistas, no haremos esto y saltaremos directamente a un ejemplo musical. El listado 5.15 muestra una mandolina (1) que es tocada por la función `recurScale()` (2). Después de definir frecuencia (3), tocar con `noteOn` (4), y avanzar el tiempo (5), la función se llama a sí misma, con una nota más baja (restar 1) y una duración más corta (90%) (7). Pero esto lo hace solo si el argumento es mayor que un valor más bajo (se detiene en la nota 40) (6). De otra forma, la función continuaría llamándose a sí misma para siempre.

Listado 5.15

```
//cadena de sonido, mandolina a la salida de audio
//(1) instrumento tipo Mandolin
Mandolin mand => dac;

//toca escala recursiva
//(2) definición de la función recurScale
fun int recurScale(int note, dur rate) {
  //(3) define la frecuencia de la nota de la mandolina
  Std.mtof(note) => mand.freq;
  //(4) toca la nota usando noteOn
  1 => mand.noteOn
  //(5) espera el tiempo indicado por rate
  rate => now;

  //solo se hace hasta llega al límite
  //(6) límite para la recursión
  if (note > 40)
  {
    //aquí está la recursión, la función se llama a sí misma
```

```

    //(7) irecurScale puede llamar a recurScale!
    recurScale(note - 1, 0.9 * rate);
  }
}

//ahora toca un par de escalas
recurScale(60, 0.5 :: second);
recurScale(67, 1.0 :: second);

```

Es así que con poco de código, eres capaz de tocar muchas y muchas notas estructuradas (no-aleatorias), usando el poder de la recursión. Podrías, por supuesto, lograr lo mismo usando un bucle for o programando explícitamente cada nota, no obstante la recursión te brinda una nueva y poderosa técnica para controlar tu sonido y música.

NOTA Deberíamos advertirte que, aunque extremadamente poderosas, las recursiones en programación pueden ser un poco peligrosas, porque siempre tienes que construir las condiciones de detención (if note > 40) en tus recursiones. De otro modo, se llamarán a sí mismas para siempre y nunca terminarán. A pesar de eso, siempre tienes los botones Remove Last Shred y Clear VM en el miniAudicle para detener cualquier proceso zombie que no quiera morir.

5.4.1 Cálculo factorial con recursión

Volvamos a la función matemática factorial, aunque pronto le agregaremos un giro musical. La función factorial (escrita en matemáticas como $N!$, pero la escribiremos aquí como una función, `factorial(N)`) calcula el producto de un entero con cualquier otro entero menor a él, hasta el entero 1. Por ejemplo, $\text{factorial}(3) = 3 \cdot 2 \cdot 1 = 6$, y $\text{factorial}(4) = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. *La función factorial tiene aplicaciones reales en estadística, cuentas, y en otras áreas.*

El número de combinaciones (permutaciones) de las cuatro letras ABCD (como ABCD, ACBD, ...) es 24 (*factorial(4)*). Observa que *factorial(4) = 4 factorial(3)*, que a su vez es *4 * factorial(2)*, y así. Esto te permite usar recursión para calcular todos los factoriales, con tal solo escribir una función, como se muestra en el listado 5.16. La función *factorial()* se llama a sí misma, a menos que su argumento sea menor a 1, en tal caso retorna 1. Entonces *factorial(4)* retorna *4 factorial(3)*, lo que retorna *3 factorial(2)*, lo que retorna *2 * factorial(1)*, lo que retorna 1. Todo esto retorna el valor final, 24.

Listado 5.16 Calculando factorial con recursión

```
fun int factorial(int x)
{
    if (x <= 1)
    {
        //cuando llegamos aquí, nuestra función termina
        return 1;
    }
    else
    {
        //la función recursiva se llama a sí misma
        return (x*factorial(x-1));
    }
}

//programa principal, llamada a factorial
<<< factorial(4) >>>;
```

5.4.2 Sonificación de la función factorial recursiva

Por diversión musical, sonifiquemos (convertir datos o procesar información en sonido) la función recursiva factorial, como se muestra en el listado 5.17. Aquí usaremos un SinOsc (1) pero añadiremos una línea dentro de la función factorial (2) para hacer una sonificación del valor actual con sonify() (3). Esa función suma la mitad de ese número a 60 (C central) (4) y toca la frecuencia asociada en tu SinOsc (5). En el programa principal sonificas las llamadas a factorial (6), entonces lo que realmente escuchas es un número de alturas graves descendientes. Escuchas una nota por cada llamada (recursiva) a factorial, seguida de la altura final más alta del resultado. Escucha atentamente; puedes escuchar que factorial(2) es igual a 2 (la misma nota en el principio y el final) y que factorial(5) resulta en un resultado que es casi demasiado alto como para ser audible. Esto demuestra, a través de sonido, que la función factorial crece rápidamente en valor para valores de argumento crecientes. Esta es una de las geniales características de la sonificación y de ChuckK.

Listado 5.17 Sonificación de la función factorial()

```
//cadena sonora, SinOsc a la salida de audio
//(1) SinOsc para que puedas escuchar el factorial
SinOsc s => dac;

//nuestra función factorial recursiva
//(2) definición de la función factorial
fun int factorial(int x)
{
    //(3) llamada la función sonify dentro de factorial
    sonify(x);
    if (x <= 1) return 1;
    else return (x * factorial(x - 1));
}
```

```

//función para sonificar números
//(4) definición de la función sonify
fun void sonify(int note) {
  //desfase sobre C central
  //(5) define la frecuencia en función de la nota
  Std.mtof(60 + (0.5 * note)) => s.freq;
  //(6) enciende el oscilador
  1.0 => s.gain;
  300 :: ms => now;
  0.0 => s.gain;
  50 :: ms => gain;
}

//
sonify(factorial(2));
second => now;
sonify(factorial(3));
second => now;
sonify(factorial(4));
second => now;
sonify(factorial(5));
second => now;

```

5.4.3 Uso de recursión para crear estructuras rítmicas

Revisemos otro ejemplo en el listado 5.18, similar a nuestros ejemplos de factorial, esta vez haciendo un patrón de redoble de tambores pero solo usando una unidad generadora tipo Impulse (1). El UGen Impulse genera un click cada vez que se lo pides. En la función `impRoll()` (2), usas el argumento `index` como tu contador para determinar cuántas veces la función tiene que llamarse a sí misma recursivamente (4), y también usas

index como retraso entre impulsos para avanzar el tiempo (3). El resultado es un redoble en constante aceleración, cuyo tempo inicial y duración total están determinados por el argumento en la llamada a la función `impRoll()` desde el programa principal (5).

Listado 5.18 Redoble de tambores recursivo usando un UGen tipo `Impulse`

```
//(1) generador de impulsos (click) al dac
Impulse imp => dac;

//(2) definición de la función impRoll
fun int impRoll(int index){
  if (index >=1)
  {
    1.0 => imp.next;
    //(3) la duración es la variable de recursión
    index :: ms => now;
    //(4) llamada recursiva de impRoll a impRoll
    return impRoll(index-1);
  }
  else {
    return 0;
  }
}

//(5) prueba con diferentes duraciones iniciales
impRoll(20);
second => now;
impRoll(50);
second => now;
impRoll(60);
second => now;
```

Ahora has visto que aunque poderosas, las funciones que pueden llamarse

a sí mismas son aún más poderosas. Puedes tocar muchas notas o sonidos y calcular resultados matemáticos complejos usando recursión. No obstante, también tienes que ser cuidadoso al diseñar funciones recursivas, para asegurarte que tienen una condición garantizada de detención.

5.5 Ejemplo: hacer acordes con funciones

Para cerrar este capítulo, queremos darte otro ejemplo musical usando funciones, dándote la habilidad de tocar diferentes tipos de acordes (múltiples notas simultáneas en armonía). Para empezar, te mostraremos una forma avanzada de declarar tu red sonora de UGens SinOsc llamado `chord[]` en la primera línea del listado 5.19 (1). Quieres que tu acorde tenga tres notas. Usas un bucle `for` para hacer `Chuck` de cada elemento de tu arreglo `chord[]` al `da(c` (2), y luego defines el `.gain` de cada `SinOsc` para que sumados sean igual a 1.0 (3).

La función `playChord()` toma un número de nota MIDI raíz como valor `tintero`, una `quality` (calidad) del acorde (mayor o menor), y una duración de tiempo durante la que el acorde sonará, llamada `howLong`. Los que sean músicos sabrán que un acorde mayor y uno menor están compuestos de tres notas: una nota raíz (4), una nota que se enucentra una tercera arriba y otra nota una quinta arriba. Tanto en acordes mayores como menores, la nota raíz y la quinta coinciden. La quinta es siete notas MIDI más alta que la raíz (5). La diferencia radica en la tercera. Una tercera mayor está cuatro notas MIDI sobre la tónica (6), con un sonido más brillante; una tercera menor está tres notas MIDI por sobre la raíz (7),

sonando más oscura. Controlas todo esto con lógica simple de if/else. También haz una prueba adicional para asegurarte que el usuario ha especificado una de las calidades del acorde, mayor o menor(mayor o menor) (8).

Listado 5.19 Tocar acordes en un arreglo de UGens SinOsc, usando una función

```
//red de sonido
//(1) tres osciladores para un acorde
SinOsc chord[3];

for (0 => int i; i < chord.cap(); i++)
{
    //(2) conecta cada elemento de nuestro arreglo al dac
    chord[i] => dac;

    //(3) ajusta ganancia para no saturar
    1.0 / chord.cap() => chord[i].gain;
}

fun void playChord(int root, string quality, dur howLong)
{
    //(4) define la nota raíz del acorde
    Std.mtof(root) => chord[0].freq

    //(5) quinta del acorde
    Std.mtof(root + 7) => chord[2].freq

    //la tercera define la calidad, mayor o menor
    //(6) acorde mayor
    if (quality == "major")
```

```

{
    Std.mtof(root+4) => chord[1].freq;
}
//(7) acorde menor
else if (quality == "minor")
{
    Std.mtof(root+3) => chord[1].freq;
}
else
{
    //(8) imprime un error en el caso de que el argumento no sea
    <<< "¡debes especificar mayor o menor!!" >>>;
}

howLong => now;

}

```

El programa principal, mostrado en el siguiente listado, es un bucle infinito que llama a la función playChord (1), generando un número aleatorio para el valor raíz y toca un acorde menor basado en la nota. Luego llama a playChord con dos acordes fijos, un acorde C menor (2) y un acorde G mayor (3).

Listado 5.20 Usando playChord

```

//programa principal, ¡usemos playChord!
while (true)
{
    //(1) toca un acorde menor en una nota aleatoria
    playChord(Std.rand2(70, 82), "minor", second/2);
    //(2) toca un acorde C menor
    playChord(60, "minor", second/2);
}

```

```
//(3) toca un acorde G mayor  
playChord(67, "major", second/2);  
}
```

Ejercicio

Cambia los parámetros de las llamadas a `playChord`, incluyendo la raíz, calidad y duración. Agrega más llamadas a `playChord` en el bucle `while`. Si eres musicalmente muy ambicioso, trata de programar un conjunto completo de acordes para una canción. Pista, "Twinkle" podría empezar así:

```
playChord(60, "major", second/2);  
playChord(60, "major", second/2);  
playChord(72, "major", second/2);  
playChord(60, "major", second/2);  
playChord(65, "major", second/2);  
playChord(65, "major", second/2);  
playChord(60, "major", second/2);
```

5.6 Resumen

En este capítulo has aprendido cómo escribir y usar tus propias funciones, incluyendo también los siguientes hechos:

- Las funciones te permiten organizar y documentar mejor tu código.
- Las funciones son declaradas por nombre (único, como las variables), tipo de retorno (`int`, `float`, `UGen`, cualquier tipo, incluso `void`), y argumentos (valores pasados para que la función opere sobre ellos).
- Las funciones que son diseñadas de manera correcta pueden ser

usadas una y otra vez entre programas.

- Las funciones se pueden llamar a sí mismas. Esto es llamado recursión.
- Las variables tienen un ámbito (scope), que es local a una función o contexto de llaves o global, visible a todo código.

Has hecho muchos sonidos y estructuras musicales interesantes con tus nuevas habilidades y conocimiento sobre funciones. Recorrer todos estos ejemplos de funciones te debería brindar un entendimiento de cómo organizar tu código en módulos y, en esencia, hacer que tus programas sean mucho más expresivos, reusables y de fácil lectura.

En el próximo capítulo, abriremos las puertas a la creación de nuevos sonidos en ChuckK, revisando una variedad de unidades generadores, que son los bloques fundamentales de la síntesis y el procesamiento de sonido.