Fundamentos: sonido, ondas y programación en ChucK

Este capítulo cubre:

- Introducción a la acústica, sonido y formas de onda
- Ondas sinusoidales y otros osciladores
- Variables
- Estructuras de control y lógica

Hasta el momento hemos mostrado por qué creemos que ChucK es el mejor lenguaje y el más poderoso para hacer todo tipo de arte y sistemas artísticos. Ahora es el momento de empezar a aprender a programar en ChucK para hacer sonido y música. Primero hablaremos de sonido en general, observando una o dos formas de onda de sonido. Discutiremos las propiedades de los sonidos en términos de volumen, altura y ruido. Aprenderás que objetos llamados osciladores son fundamentales para la física, el sonido y la música, y harás música con los osciladores incluidos en ChucK. Aprenderás:

- Cómo los datos son manejados y manipulados por ChucK mediante el uso de variables.
- Sobre los mecanismos de temporización incluidos en ChucK. La forma en que maneja el tiempo es uno de los aspectos que lo hace diferente de los otros lenguajes de programación, y que lo hace tan bueno para programar música, sonido y arte basado en el tiempo.
- Sobre el control del flujo de tus programas mediante el uso de variables lógicas y pruebas y bucles.

¡Para el final del capítulo habrás escrito tu primera composición con ChucK!

1.1 Ondas de sonido y formas de onda

Empezaremos discutiendo sobre la física y la naturaleza del sonido. El sonido consiste de fluctuaciones de alta y baja presión (ondas) de aire que son causadas por uno o más objetos vibrando. Las ondas de sonido luego se propagan a través del aire, quizás rebotando contra las paredes y otras superficies, para finalmente alcanzar tus oídos o un micrófono. La gente que trabaja con sonido generalmente grafica las formas de onda (amplitud de la presión de aire, o voltaje de un micrófono, como una función del tiempo). La figura 1.1 muestra un gráfico de valores de onda como una función del tiempo de un hombre diciendo la palabra "see".

Algunos aspectos son obvios en este gráfico. La consonante ruidosa "sss" en la primera mitad del sonido cambia rápidamente a una distinta estructura para la vocal "eee" en la segunda mitad. Si haces zoom a la zona de transición, más aspectos se hacen obvios, como muestra la figura 1.2.

La "sss" se ve prácticamente igual, todavía ruidosa con muchas ondulaciones en la forma de onda, pero el sonido "eee" rápidamente entra a lo que se llama oscilación periódica, donde una forma básica se repite una y otra vez. Podrán haber pequeñas desviaciones debido a ruido, temblores en la voz en sí misma, entre otros factores, pero la oscilación generalmente se repite una y otra vez. Esta es una característica de los sonidos que tienen altura. La altura es tu percepción de la frecuencia, de grave a alta. Es lo que cantas o silbas de una melodía de una canción, y es

lo que los músicos anotan y discuten en términos de nombres de notas y números. Las teclas de un piano son orientadas por altura de izquierda (grave) a derecha (alto).

Casi todo lo que te da una sensación de altura posee una oscilación periódica, y cualquier objeto que oscile periódicamente producirá un sonido de una cierta altura (dentro de un rango de frecuencias). La región marcada como T en la figura 1.2 es el periodo de la oscilación (usualmente medido en segundos), y 1/T es la frecuencia de la oscilación en ciclos por segundo. Así que en el caso del sonido "eee", si el periodo T mide 6.6 milisegundos (un milisegundo es una milésima de un segundo), entonces la frecuencia de oscilación será 1/0.0066 = 150 ciclos por segundo, o 150 Hz (la unidad de frecuencia nombrada en honor al físico Heinrich Hertz).

Si repites ese único periodo del sonido "eee" una y otra vez, repitiéndolo en un punto conveniente como en alguna cima principal, podrás sintetizar un sonido "eee" de la duración que quieras. Si reproduces ese sonido un poco más rápido, la altura subirá. Tócalo a una velocidad menor y obtendrás una altura menor. El sonido sintético será aburrido, porque al seleccionar un solo periodo y repetirlo una y otra vez de forma exacta, habrás eliminado todo el ruido, pequeñas desviaciones en altura, y otros aspectos que hacen que el original suene natural. Esta es una razón que explica por qué los sonidos de voz sintetizada suenan, ah, sintéticos.

Mucha de la historia de la música electrónica, y la historia de la música en general, y realmente de gran parte de la física, se centra en la noción de oscilación. Usaremos muchos sabores de osciladores incluidos en ChucK en las siguientes secciones, pero por ahora observaremos una de las formas fundamentales de oscilación en la naturaleza y el sonido: la onda sinusoidal. Pronto usarás el oscilador de onda sinusoidal de ChucK para

escribir tu primer programa que emite una nota musical. ¿Entonces, qué es una onda siusoidal? Una onda sinusoidal cae y decae de forma suave y periódica, como se muesrta (por un par de periodos) en la figura 1.3.

El círculo de la izquierda muestra una forma en que se pueden graficar, explicar y generar las ondas sinusoidales. Si rotas el círculo en dirección contraria a las agujas del reloj a una tasa constante, digamos N rotaciones por segundo, entonces la altura del punto que empieza en el lado izquierdo del círculo (el lado izquierdo del gráfico) traaz una onda sinusoidal mostrada hacia la derecha, según el tiempo avanza de izquierda a derecha. Una manera de entender esto es imaginarse el punto en el borde del círculo como un lápiz que si tomas un pedazo de papel de izquierda a derecha mientras el círculo avanza, dibujará la onda sinusoidal. Si N fueran cinco ciclos por segundo, entonces el círculo y la onda sinusoidal completarán cinco ciclos en un segundo.

Puedes haber escuchado sobre funciones sinusoidales si has estudiado trigonometría, pero como hicimos notar antes, las ondas sinusoidales pueden ser encontradas en muchos lugares de la naturaleza y de las matemáticas. La electricidad en las tomas de corriente alterna tienen un patrón sinusoidal, porque los generadores en las plantas de generación eléctrica rotan en círculos. Pero no solo rotando objetos se pueden generar ondas sinusoidales. Un péndulo describe una onda sinusoidal de desplazamiento mientras oscila hacia adelante y atrás. Un circuito eléctrico que contiene un inductor y un capacitor oscila de forma sinusoidal. Acústicamente, la resonancia única del aire dentro de una botella oscila de forma sinusoidal. Así que como puedes ver, las ondas sinusoidales están presentes en todas partes y son una característica importante del mundo real.

Otras formas de onda simples que son comunes en la naturaleza y en la electrónica incluyen las ondas triangulares, diente de sierra y cuadradas. La figura 1.4 muestra dos ciclos cada una de onda sinusoidal, triangular, diente de sierra y cuadrada.

Las ondas sinusoidales están en el corazón del análisis de sonido y también de la síntesis. Ondas periódicas más complejas pueden ser construidas a partir de ondas sinusoidales. Puedes construir cualquier sonido posible sumanod ondas sinusoidales de distintas frecuencias, fase (retraso) y amplitudes (volumen). Las ondas sinusoidales son los ladrillos fundamentales de construcción de lo que llamamos visión espectral del sonido, donde en vez de revisar los pliegues individuales de la onda, revisas los componentes sinusoidales que conforman esta forma de onda.La figura 1.5 muestra que una onda diente de sierra puede ser construida sumando ondas sinusoidales en múltiplos enteros (1, 2, 3, ...) de alguna frecuencia fundamental. Estos componentes son llamadas armónicos. Las ondas triangular y cuadrada pueden ser construidas con frecuencias armónicas impares (1, 3, 5)de ondas sinusoidales a distintas amplitudes.

La onda diente de sierra también tiene una analogía física en el mundo de los instrumentos musicales, porque cuando una cuerda arqueada oscila, el arco alternadamente se apega a la cuerda, arrastrándola consigo, y luego se desliza, dejando que la cuerda rápidamente vuelva a su posición original. Esto crea un movimiento tipo diente de sierra, tal como lo observó hace mucho Herman von Helmholtz (1821 - 1894). Helmholtz nos brindó algunas de las teorías acústicas y experimentos más tempranos de la descripción espectral de instrumentos y sonidos. Una onda diente de sierra puede ser un buen punto de partida en la síntesis de un violín, y lo realizaremos pronto.

Físicamente, un clarinete tocando su nota más grave oscila muy parecidamente a una onda cuadrada, así que puedes construir una síntesis muy simple que suene como un clarinete usando una onda cuadrada y un generador de envolvente (más información dentro de poco). ¡Pero primero, programemos!

1.2 Tus primeros programas de ChucK

Es tiempo de empezar a aprender a programar. En este punto, deberías haber leído las intrucciones del apéndice A sobre cómo instalar miniAudicle y la funcionalidad básica del entorno de programación. Una vez que ha sido instalado, abre miniAudicle y te guiaremos para escribir tus primeros programas. La figura 1.6 muestra al miniAudicle listo para ejecutar código en ChucK.

Existen tres ventanas principales en el miniAudicle. La ventana prinicpal que ves en la parte superior de la figura 1.6 se llama inicialmente Untitled (sin título), y es la ventana en la que escribes tus programas. Puedes cambiar el nombre de la ventana, lo que haremos dentro de poco cuando escribamos y grabemos nuestro primer programa. La ventana de la esquina inferior derecha es el monitor de la máquina virtual (VM, por Virtual Machine), que muestra el estado del servidor de la VM mientras tu código en ChucK es ejecutado. Una vez iniciada, la VM está siempre ahí corriendo, esperando cosas que hacer.

Cuando haces click en Start Virtual Machine (iniciar la máquina virtual), crea una ventana llamada Console Monitor (monitor de consola), mostrada en la esquina inferior izquierda de la figura 1.6. Aquí es donde recibirás mensajes del computador para ayudarte con tu programa. Si hay algún

error en el código, aquí es donde el computador te dirá que existen problemas y específicamente dónde (en cuáles líneas) ocurren. También puedes imprimir mensajes en el Monitor Console a ti mismo desde tu código en ejecución para ayudarte a corregir errores en tu programa o para mantenerte informado sobre lo que está pasando.

1.2.1 Tu primer programa: "Hello World"

En casi todos los lenguajes, el primer programa que escribes es "Hello World" (Hola Mundo), así que empezaremos mostrándote cómo lograr esto en ChucK. Dirígite a la ventana llamada Untitled, grábala con el nombre helloworld.ck usando el menú File (Archivo), y luego escribe esta línea:

```
<<< "Hello World" >>>;
```

No te olvides de incluir el punto y coma, que es importante para señalar el término de cada línea de código en ChucK.

Luego, asegúrate de que tu VM esté corriendo (verás el tiempo transcurrido: el valor aumenta constantemente), y luego haz click en Add Shred (Añadir Shred) en la ventana principal, que ahora se llama helloworld.ck. De forma instantánea verás que "Hello World" aparece en el Console Monitor. ¡Felicidades, has ejecutado correctamente tu primer programa en ChucK! Haciendo click en el botón Add Shred le dice ChucK que ejecute el código "Hello World" que escribiste en la ventana principal. Harás click en este botón cada vez que quieras ejecutar un programa en ChucK.

Aunque siempre estarás escribiendo y ejecutando código desde la ventana

principal, desde ahora solo mostraremos el código como texto, con etiquetas y comentarios. No mostraremos la ventana principal completa con los botones de control, pero mostraremos partes de la figura 1.6 en el futuro para señalar algunos de los controles. Todas las ventanas pueden ser movidas en tu escritorio según prefieras, tal como muchos de los programas con los que estás acostumbrados a trabajar con en tu computador.

Tal como fue discutido anteriormente, es útil para ti como programador imprimir al Console Monitor para efectos de detectar y corregir errores y como compositor para saber en qué parte de la canción va la ejecución. Desde este primer ejemplo sabes que poner ítemes dentro de <<< >>> te permite imprimirlos. Poneros entre comillas dentro de los triples <<< >>> los imprime directamente. Te daremos técnicas más avanzadas pra imprimir otras cosas a lo largo de este capítulo.

1.2.2 Tu primer programa sonoro: "Hello Sine!"

Como ChucK es un lenguaje orientado a sonido, y nuestra meta es enseñarte a programar a través de la escritura de composiciones, necesitaremos llegar mucho más allá de imprimir texto en la ventana Console Monitor. Necesitas aprender cómo generar sonido. Es así que tu segundo programa se llamará "Hello Sine!" (Hola onda sinusoidal). Como una onda sinusoidal es periódica, produce un sonido que puedes identificar con una altura. Así que piensa en este primer sonido de onda sinusoidal como la nota inicial de tu primera canción, "Twinkle, Twinkle, Little Star".

Cuando ingresas el código de la sección 1.1, asegúrate de grabarlo como un nuevo archivo llamado HelloSine.ck. Te acostumbrarás a grabar tus

programas con nombres que tnengan sentido. Cuando modifiques de gran forma tus programas, es a menudo grabarlos como un nuevo archivo, dándoles un nombre relacionado, como HelloSine2.ck.

También nota que mientras escribes el programa del listado 1.1 en el editor de miniAudicle, algunas de las palabras mágicamente cambian de color. Esto ocurre porque el miniAudicle reconoce estas palabras reservadas y las colorea según su tipo. Aprenderemos más de esto más adelante.

Listado 1.1 Hello Sine!

```
//Conecta una onda sinusoidal
SinOsc s => dac;
//Define la ganancia del oscilador como 0.6
0.6 => s.gain;
//Define la frecuencia del oscilador como 220
220 => s.freq;
//Permite que toque por un segundo
second => now;
```

Diseccionemos este programa línea a línea. En la primera línea, construyes tu primera cadena de señal de sonido, también llamada patch, sobre la que hablaremos pronto. SinOsc es llamado una unidad generadora (UGen), un objeto que genera el sonido de una onda sinusoidal. El nombre que le das al oscilador es s, lo que también es llamado en programación como el nombre de una variable porque su valor puede cambiar. Es también una variable en el sentido de que podrías haber escogido cualquier otro nombre como t o mionda. Tu como programador eliges cómo nombras los objetos que creas. Por simplicidad, hemos escogido s.

La siguiente parte importante es el símbolo =>. Observa que se parece a

una flecha. Este es el operador ChucK, y ChucK es el nombre del lenguaje completo. Está diseñado para mostrar cómo una señal fluye de un lado de la flecha al otro. Cuando lo usas, dices que estás ChucKing lo que sea que está al lado izquierdo de la flecha a lo que sea que está al lado derecho. Así que ahora, tienes un SinOsc que has llamado s y quieres enviar el sonido a tus parlantes o audífonos. Para esto usas un objeto especial llamado un conversor análogo-digital (DAC, por digital-to-analog converter), el que está conectado dentro de tu computador a la salida de tu tarjeta de sonido (parlantes, audífonos, o similar).

Entonces, por medio de ChucKing un SinOsc llamado s al dac, has construido una onda sinusoidal que serás capaz de escuchar a través de tus parlantes. Otro término para conectar tu unidad generadora tipo oscilador sinusoidal al dac es hacer un patch, que proviene de los viejos tiempos de los sintetizadores análogos donde los componentes generadores y procesadores de sonido eran conectado usando cables patch, o cables físicos con enchufes para hacer las conexiones.

Ahora que has creado una cadena sonora capaz de producir sonido, es necesario saber que existen tres aspectos principales de este programa para crear sonido: cuánto volumen, cuál frecuencia (o altura), y cuán largo. En la línea 2, defines cuánto volumen. Todos los objetos generadores y procesadores de sonido en ChucK tienen un método .gain que puede ser ajustado a cualquier número, pero está usualmente en el rango entre 0.0 y 1.0. Observa que en este código has definido el volumen a .6 por medio de su Chucking a s.gain.

En la línea 3, trabajas con la frecuencia. Así es como eres capaz de componer y determinar cuál altura (frecuencia específica) quieres que tenga tu SinOsc. El método .freq de tu SinOsc acepta un número en Hertz

entre 0 y 20,000, que cubre, y sobreestima por debajo, el rango de frecuencias de escucha humana. Notar en que en este ejemplo defines la frecuencia por medio de Chucking 220 a s.freq. Esto es la altura de la primera nota de tu canción "Twinkle". Conversaremos más en el siguiente capítulo sobre cómo determinar frecuencias para alturas musicales particulares.

En la línea 4, determinas cuán larga será la ejecución de tu síntesis, o la duración de tu primera nota. Por ahora quieres que tu sonido se ejecute por un segundo, lo que logras por medio de ChucKing una duración de 1 segundo a un objeto llamado now. Hacer ChucKing de second a now le indica a ChucK que quieres esperar 1 segundo mientras el sonido es sintetizado. Si no tuvieras esta línea que manipula el tiempo, no escucharías ningún sonido, porque el programa terminaría justo ahí.

Unidades de tiempo y duración en ChucK

> Podrías haber reproducido tu onda sinusoidal durante cualquier cantidad de tiempo por medio de ChucKing una diferente duración a now. Por ejemplo, si cambias la línea 4 en el listado 1.1 a:

```
2.0 :: second => now;
```

> el sonido se reproduciría por 2 segundos. Los dos puntos le indican a ChucK que quieres hablar sobre unidades de tiempo (duration en segundos), entonces 2 :: second es 2 segundos, 0.1 :: second es 1/10 de un segundo, y así. También observa que pueden haber espacios (o no) alrededor del par de dos puntos. Discutiremos pronto más sobre otras unidades de tiempo/duración en ChucK.

Uno de los aspectos más poderosos de ChucK es que como programador tienes absoluto control del tiempo. Pero también estas obligado a controlar el tiempo para hacer sonidos. Discutiremos tiempo y el objeto now en más detalle en la sección 1.4.

1.2.3 Ahora hagamos música

Tocar un única onda sinusoidal es una gran cosa, pero ahora expandamos nuestro programa para que pueda tocar cuatro notas. El siguiente paso es construir nuestra partitura de alturas y duraciones que queremos que nuestro SinOsc haga. Primero, añade un comentario al principio del programa, para ayudar a documentarlo. Luego, extiende tu previo programa como mostramos a continuación.

Listado 1.2 Música con onda sinusoidal

```
//Bloque 1
/* Música con onda sinusoidal
por un programador de ChucK
Enero 2025
*/

//Bloque 2
//Conecta una onda sinusoidal al dac
SinOsc s => dac;

//Bloque 3
//Toca una nota
//Define la ganancia como 1.0 y la frecuencia como 220 Hz.
//Deja que se ejecute por 0.3 segundos haciendo ChucKing a now
220 => s.freq;
1.0 => s.gain;
```

```
0.3 :: second => now;

//Bloque 4
//Calla tu oscilador por 0.3 segundos para separarlo de la sigu
0.0 => s.gain;
0.3 :: second => now;

//Bloque 5
//Toca una nota, con la misma altura
//Repite el proceso de los bloques 1 y 2
1.0 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;

//Bloque 6
//Toca dos notas adicionales, más altas, menor volumen
//Repite el mismo patrón de los bloques 1, 3 y 4
//pero con una diferente frecuencia y volumen
330 => s.freq;
0.3 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;

0.3 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;
```

Observa que estás creando tu primera nota individual definiendo la

ganancia del oscilador de onda sinusoidal conectado al dac a 1.0 y luego a 0.0, dejando que tu patch se ejecuta por un pequeño lapso de tiempo por medio de ChucKing 0.3 secounds a now. Repites la misma nota en el bloque 5. Tienes que hacer el silencio entre notas (definiendo .gain como 0.0) para que esas dos primeras notas repetidas no suenen como una sola nota larga. En el bloque 6, repites de forma completa la secuencia de dos notas pero con una altura un poco mayor (el segundo "twinkle" de tu canción) y a un volumen más bajo (la ganancia de un oscilador está correlacionada con el volumen, a mayor ganancia mayor volumen). Puedes problamente imaginarte creando una composición completa, hilando comandos como estos. Pero eso es mucho trabajo, y para el final de este capítulo te habremos mostrado muchas técnicas para crear notas como esta de una forma más concisa y poderosa.

Chuck es un lenguaje de programación de verdad

> Para aquellos de ustedes que han trabajado con softwares musicales, como GarageBand pra Mac o FruityLoops para PC u otros programas, pareciera ser mucho trabajo el escribir tanto para producir solo unos cuantos tonos sinusoidales. ¿Recuerdas cuando mencionamos en la introducción ("¿Qué es ChucK? ¿Por qué es diferente?") que como ChucK está basado en texto, teclear es parte del trato? Pero como es posible y se alienta copiar y pegar, y en verdad la cantidad de escritura requerida en ChucK es mucho menor que cualquier otro lenguaje basado en texto, incluso en los que fueron diseñados para fines musicales.
>
> Por ahora, para darte un vistazo del poder de ChucK, puedes escribir (o copiar y pegar) y correr este programa corto. No te preocues si hay cosas que no entiendes; solo ten cuidado de escribirlo tal como lo ves

> aquí:

```
Impulse imp => ResonZ filt => NRev rev => dac;
0.04 => rev.mix;
100.0 => filt.Q => filt.gain;

while (1) {
  Std.mtof(Math.random2(60,84)) => filt.freq;
  1.0 => imp.next;
  100 :: ms => now;
}
```

> Una vez que hayas escrito el código, grábalo como WowExample.ck (o cualquier otro nombre que prefieras), y haz click en el botón Add Shred (figura 1.6, esquina superior izquierda). Escucharás un flujo constante de pops afinados a frecuencias aleatorias. Mucho sonido puede ser logrado a partir de poco código, una vez que conoces las poderosas características de este lenguaje. Puedes hacer click de nuevo en Add Shred y escuchar aún más notas. De hecho, puedes hacer click muchas veces en Add Shred y escuchar aún más sonidos.
>
> Cuando estés listo y quieras detener el sonido, haz click en el botón Clear Virtual Machine (Vaciar la máquina virtual) en la esquina superior derecha de la ventana de miniAudicle (figura 1.6).
>
> A lo largo de los siguientes ejemplos y los próximos capítulos, estarás construyendo tu conocimiento básico sobre ChucK, permitiéndote hacer programas cada vez más poderosos. Si a veces sientes que no estás obteniendo demasiado de tus líneas de código escritas, ten paciencia, porque al final de este capítulo habrás aprendido nuevos aspectos de ChucK que te permitirán hacer mucho más, incluso horas

> de música, con muy poco código adicional.

1.2.4 Probando nuevas formas de onda

Como estás listo para empezar a componer usando un oscilador, queremos permitirte que uses más que solo un objeto SinOsc para hacer música. Como lo discutimos en la seccción 1.1, es posible hacer sonido usando distintos tipos de formas de onda. Prueba las tres formas de onda mostradas en la figura 1.7 en el programa del listado 1.2 cambiando el SinOsc en el bloque 2 de código por uno de los siguientes: onda cuadrada (SqrOsc), onda triangular (TriOsc), onda diente de sierra (SawOsc).

Haciendo pruebas y corrigiendo errores

> Cada vez que haces un cambio en tu programa, harás click en el botón Add Shred para indicarle a ChucK que corra el código nuevo. Si hay un error en tu escritura, ChucK lo imprimirá en la ventana Console Monitor, usualmente acompañado del número de línea donde encontró el error. Este número de línea estará marcado con color rojo en la ventana principal de miniAudicle donde escribiste tu código. Vuelve y corrige tu error, y vuelve a correr el código haciendo click en Add Shred hasta que todo funcione. Si todavía no funciona o hace sentido, reescribe el código desde cero. ¡Así es como los programadores lo hacen!
>
> Además, en cualquier momento puedes usar el ítem del menú Save As para grabar un nuevo archivo de código ChucK con un nuevo nombre de archivo. Esto será importante más adelante a medida que escribes más y más programas.

Algunos de los elementos más importantes en programación son los comentarios que el autor hace mientras crea el código. Como mostramos en las primeras líneas del listado 1.2 y mostraremos de nuevo aquí, los comentarios se pueden hacer de dos maneras: la primera es con un conjunto de barra oblicuas y astericos, / *y* /, y la segunda es con dos barras oblicuas //. En el método 2, cualquier caracter en la misma línea a continuación de las dos barras será ignorado por ChucK. El método 1 es para múltiples líneas, donde todas las líneas entre el par de caracteres será ignorado por el compilador.

```
//Bloque 1
/* Música con onda sinusoidal
por un programador de ChucK
Enero 2025
*/

//Bloque 2
//Conecta una onda sinusoidal al dac
SinOsc s => dac;

//Bloque 3
//Toca una nota
//Define la ganancia como 1.0 y la frecuencia como 220 Hz.
//Deja que se ejecute por 0.3 segundos haciendo ChucKing a now
220 => s.freq;
```

Buenas prácticas de programación

> Al principio, deberías hacerte el hábito de comentar cada línea (o la moyoría de las líneas, o al menos cada bloque) de código con anotaciones para ti mismo y para futuros colaboradores. Además,

> deberías siempre escribir tu nombre y fecha al principio de cada programa que escribes. A medida que te vayas convirtiendo en un programador más avanzado y también consultes los programas de otros, aprenderás cómo comentar de manera más efectiva, dónde los comentarios son necesarios, y qué decir en ellos.

El ejemplo del listado 1.3 reune todos los conceptos discutidos en la sección 1.2. El código está bien documentado (comentado) y también imprime algo en la salida 1 de la consola al principio y luego toca dos notas de música.

En el último bloque 2, te mostramos cómo puedes usar el comentario multilínea /.../ para hacer que una sección de código no sea ejecutada, porque ChucK asume que este texto es un comentario y no código. Esto puede ser extremadamente útil para reparar errores en tu código o para omitir una parte de tu composición para que así puedas trabajar con secciones posteriores.

```
//Autor: equipo ChucK
//Fecha: la fecha de hoy

//Haz una cadena de sonido (patch)
SinOsc s => dac;

//Imprime el nombre del programa
<<< "Hello Sine!" >>>;

//Define los parámetros para tocar una nota
//Define el volumen como 0.6
.6 => s.gain;
//Define la frecuencia como 220
220.0 => s.freq;
```

```
//Toca por un segundo
second => now;

//Define el volumen como 0.5
0.5 => s.gain;
//Define la frecuencia como 440
440 => s.freq;
//Ahora toca por dos segundos
2 :: second => now;

//Comenta esta tercera nota por ahora
/*
0.3 => s.gain;
330 => s.freq;
3 :: second => now;
*/
```

Ahora sabes cómo hacer sonido con osciladores, cómo cambiar la altura usando .freq y el volumen usando .gain y cómo controalr las duraciones de tus notas usando now. Pero hay otras cosas que necesitas saber para hacer mejor música.

1.3 Tipos de datos y variables

Los programas que hemo visto hasta el momento emiten sonido, pero sería muy díficil pensar en hacer una canción entera usando este tipo de programación literal para cada ganancia, frecuencia y tiempo. El siguiente concepto a aprender es cómo hacer que tu programa sea fácil de cambiar, dadas las decisiones que quisieras tomar como compositor o incluso decisiones que el programa mismo quiera tomar.

Una clave para hacer esto es usar variables. En ChucK y muchos otros

lenguajes de programación, cada variable tiene un tipo de datos particular, lo que significa que puede almacenar solo un tupo específico de valor o colección de valores. Tal como nuestros ejemplos en el listado 1.2, donde una variable s contenía nuestra SinOsc, puedes declarar variables para almacenar un número, un tiempo, o incluso palabras. Pero cada una puede almacenar solo un tipo de datos.

Un tipo de datos muy importante de ChucK está diseñado para almacenar números enteros (int). Un número entero es un tipo especial de número que no tiene una parte decimal; o sea no necesita un punto decimal. En ChucK, una variable entera puede almacenar cualquier número en el rango entre -2,147,483,648 a +2,147,483,647. Puedes declarar e inicializar (asignarle un valor a) una variable entera como se muestra en el listado 1.4. Al declarar una nueva variable se crea espacio en el computador para almacenar esa variable y registra el nuevo nombre declarado y tipo de la variable, para que puedas referirte a ella por su nombre más adelante. Inicializar una variable le da un valor. En el listado 1.4, justo después de que declaras la nueva variable myPitch, almacenas el valor 220 en ella por medio de ChucKing.

Buenas prácticas de programación; nombres de variables

> Como tú eres el programador, puedes escoger cualquier nombre que quieras para tus variables, pero siempre es mejor escoger nombres que tengan sentido. Nombres como myPitch, myVolume y fastTempo tienen mucho mayor significado que x,j y z. La gente que lee tu código puede ver estos nombres y hacerse una idea de su propósito. Más adelante verás que algunas variables son temporales, usadas de forma rápida en el código justo después de ser creadas y nunca vueltas a usar. Para aquellas variables, i, temp, x y otros nombres de

> ese estilo son adecuados. Pero para la mayoría de las variables, un buen nombre (no demasiado largo pero definitivamente no demasiado corto) es importante.

Listado 1.4 Definiendo y usando una variable entera

```
//declara un entero
int myPitch;

//almacena una variable
220 => myPitch;

//imprime su valor
<<< myPitch >>>;
```

Listado 1.5 Inicializando y declarando una variable entera al mismo tiempo

```
//otra manera de inicializar enteros
//almacena 220 en la recién declarada variable myPitch
220 => int myPitch;

//imprime su valor
<<< myPitch >>>;
```

En muchas ocasiones vas a querer cambiar el valor de una variable - esa es una de las razones por la que se llaman variables. Y hay veces en las que vas a querer usar el valor de una variable para inicializar una nueva variable. Puedes definir el valor de una variable en relación a otra para usarlas como frecuencias de un acorde musical o escala. ADemás, derivar nuevas variables a partir de otras hace que tu programa sea de más fácil lectura y modificación. Primero aprenderás a usar matemáticas para

cambiar los valores de las variables y derivar nuevas variables a partir de otras en los listados 1.6 y 1.7; luego usarás estas herramientas para tocar música en el listado 1.8. Como los enteros son números, puedes realizar todo tipo de aritmética con ellos, como mostramos en el siguiente listado. Esto cobrará mayor importancia más adelante cuando quieras tocar más y más notas.

Listado 1.6 Matemáticas con enteros

```
//aritmética con enteros
220 => int myPitch;

//sumar o restar
myPitch + myPitch - 100 => int anotherPitch;

//multiplicación
2 * myPitch => int higherPitch;

//división
myPitch / 2 => int lowerPitch;

//imprimir los valores
<<< myPitch, anotherPitch, higherPitch, lowerPitch >>>;
```

El código imprime lo siguiente en la ventana Console Monitor:

```
220 330 440 110
```

Ejercicio

> Revisa las operaciones matemáticas sobre las variables del listado 1.6 usando lápiz y papel. Comprueba que tu resultado sea los mismos

> números. Escribe este código en ChucK para comprobar tus respuestas. ¿Cuál fue más fácil? ¡Ahora puedes usar ChucK como una calculadora!

El siguiente listado muestra unos pocos métodos atajo para realizar aritmética en variables para cambiar sus valores en el mismo lugar. Añadiendo un operador matemático al principio del operador ChucK =>, el cálculo matemático es hecho en la variable y almacenado en la misma variable.

Listado 1.7 Atajos matemáticos: multiplicación, resta, y ChucK en un mismo paso

```
//versión larga
int myPitch;
220 => myPitch;

//multiplicar por dos
2 * myPitch => myPitch;

//resta
myPitch - 110 => myPitch;

<<< myPitch >>>;
```

```
//versión corta
220 => int myPitch;

//multiplicar por dos
2 *=> myPitch;

//resta
```

```
110 -=> myPitch;

<<< myPitch >>>;
```

Ambos ejemplos imprimen lo siguiente en la ventana Console Monitor:

```
330 :(int)
```

Usemos este conocimiento adquirido sobre variables enteras para tocar un par de notas, como se muestra en el siguiente listado. Aquí almacenamos la altura en una variable entera, y luego creas dos enteros llamados onGain y offGain para almacenar los valores 1 y 0 para poder prender y apagar el oscilador.

Listado 1.8 Tocando notas con variables enteras

```
/* Música sinusoidal con variables entersa
por un programador de ChucK
enero 2025
*/

SinOsc s => dac;

//Bloque 1
//declara e inicliazia una variable entera llamada myPitch
220 => int myPitch;

//Bloque 2
//declara e inicializa dos variables enteras para controlar ga
1 => int onGain;
0 => int offGain;
```

```
//Bloque 3
//toca una nota usando tus nuevas variables enteras
myPitch => s.freq;
onGain => s.gain;
0.3 :: second =>  now;

//Bloque 4
//apaga el sonido del oscilador para separar las dos notas
offGain => s.gain;
0.3 :: second => now;

//Bloque 5
//multiplica por dos la altura
2 *=> myPitch;

//Bloque 6
//define la frecuencia del oscilador y lo enciende, para empez
myPitch => s.freq;
onGain => s.gain;
0.3 :: second =>  now;

//Bloque 4
//apaga el sonido del oscilador para terminar la segunda nota
offGain => s.gain;
0.3 :: second => now;
```

Creando variables enteras en los bloques 1 y 2, puedes usarlas en vez de números para definir los parámetros de la onda sinusoidal en los bloques 3, 4 y 6. Como son variables, puedes cambiar su valor en cualquier momento, como se ve en la elegancia del bloque 5, donde multiplicas por 2 la altura. Definir la frecuencia de la onda sinusoidal con este nuevo valor hace que la altura aumente.

El tamaño del cambio en la frecuencia (el doble) es conocido como un intervalo musical de una octava hacia arriba, por razones algo oscuras. Una octava es el doble (hacia arriba) o la mitad (hacia abajo) de la frecuencia. Para aquellos familiares con instrumentos con teclado, una octava es el espacio entre dos notas C (C central a la octava superior, por ejemplo), dos notas A, dos notas G, y así. Si tocas una cuerda abierta en una guitarra o cualquier otro instrumento de cuerda y luego pones el dedo en el traste 12 y tocas la misma cuerda, escucharás una diferencia de una octava (duplicando la frecuencia, lo que no por coincidencia corresponde a la mitad de la cuerda).

Puedes tocar algunas notas usando variables enteras, ¿pero qué pasa si quieres tocar un intervalo distinto, como en caso de la canción "Twinkle", donde quieres tocar dos notas de 220 Hz y luego dos notas de 330 Hz? Puedes darte cuenta que podrías multiplicar 220 por 1.5 y eso resultaría en 330.0, pero ChucK no te permitiría hacerlo porque no puedes usar fracciones o números con decimales cuando estás manejando enteros. ¿Y qué ocurre con las siguientes dos notas ("little") de la canción? Estas notas necesitan tener una frecuencia de 369.994 (lo que podrías quizás aproximar a 370), pero qué pasa si te quieres poner creativo y cambiar la melodía un poco más? Podrías querer tocar la palabra "little" en F# y G#, cuyas frecuencias corresponden a 369.994 y 415.305 Hz. Discutiremos más sobre altura y frecuencia en el siguiente capítulo, pero necesitas saber que las variables enteras solo pueden almacenar números sin parte fraccional (no se permiten números decimales).

Afortunadamente, existe otro tipo de datos en ChucK llamado float. Los números de punto flotante son números que tienen un punto decimal (parte fraccional) y por lo tanto pueden ser más precisos,. Como te puedes imaginar, estos números son almacenados y operados de forma distinta

dentro del computador; por lo que existe un tipo de datos separado para ellos. Puedes declarar e inicializar variables float tal como lo has hecho con los enteros. Toda la aritmética y operadores que hemos visto también aplican para los float. El listado 1.9 muestra cómo puedes declarar e inicializar un float para almacenar tu primera frecuencia "twinkle". Puedes operar matemáticamente sobre ella, multiplicando por 1.5 para almacenarla en una nueva variable declarada de nombre twinkle2. De forma similar, puedes derivar tus alturas para "little" multiplicando twinkle o twinkle2 por los números de punto flotante adecuados.

Listado 1.9 Uso y manipulación de variables float

```
//bloque 1
//declara e inicializa un float para almacenar la altura de tw
220 => float twinkle;

//cálculo para derivar la altura de twinkle2 a partir de twink
1.5 * twinkle => float twinkle2;

//más cálculos para derivar la altura de lit a partir del twin
1.6818 * twinkle => flloat lit;

//más cálculos para derivar la altura de tle a partir del twin
1.2585 * twinkle2 => float tle;

//nueva altura una octava más arriba que twinkle
2 * twinkle = float octave;

<<< twinkle, twinkle2, lit, tle, octave>>>;
```

Este codigo produce la siguiente salida en la consola:

```
220.000000 330.000000 369.996000 415.305000 440.000000
```

La belleza de este código radica en que basta con cambiar la inicialización de tu primer twinkle, para que todas las otras alturas cambien. Como las variables son usadas para definir alturas, si cambias el bloque 1 a

```
110 => float twinkle;
```

entonces todas las alturas bajarán una octava. Si defines el primer twinkle como 261.616 Hz tocaría "Twinkle" en la escala de C mayor. Cambiar una melodía o canción a una escala diferente es llamado transposición. Puedes inicializar la variable inicial twinkle a cualquier número razonable y todos los otros cambiarán proporcionalmente. Así, la canción "Twinkle" estaría intacta pero en una escala musicak distinta. ¡El poder de la programación!

Ten en cuenta que ha usaste floats cuando definiste la ganancia de tu UGen SinOsc en el listado 1.2 (primero 1.0 y luego 0.3 para hacerlo menos fuerte). Revisaremos nuevamente este ejemplo en el siguiente listado para reenforzar el uso de variables float. La habilidad de convertir el volumen y la frecuencia en variables será un elemento importante al momento de hacer composiciones expresivas, como pronto verás.

Listado 1.10 Twinkle con variables float

```
/* Música Twinkle Sinuosidal con variables float
por un programador de ChucK
enero 2025
*/

SinOsc s => dac;
```

```
//Puedes usar una variable float para twinkle y usarla para ca
220.0 => float twinkle;
1.6818 * twinkle => float little;

1 => int onGain;
0 => int offGain;

//Toca una nota
//Prende la nota twinkle (hace que la ganancia sea onGain y ava
twinkle => s.freq;
onGain => s.gain;
0.3 :: second => now;
//Apaga la nota (hace que la ganancia sea offGain y avanza el t
offGain => s.gain;
0.3 :: second => now;

//Modifica la altura de twinkle usando matemática para tocar el
1.5 *=> twinkle;

//Toca la otra nota del segundo "twinkle"
twinkle => s.freq;
onGain => s.gain;
0.3 :: second => now;
offGain => s.gain;
0.3 :: second => now;

//Toca una nota de "little"
little => s.freq;
onGain => s.gain;
0.3 :: second => now;
offGain => s.gain;
0.3 :: second => now;
```

Ahora qhas usado variables (tanto int como float) para hacer tu código más

felxible y legible. ¿Pero puedes hacer algo con esas líneas con números que controlan el tiempo? Puedes hacer una variable float e inicializarla a 0.3 y luego usarla de esta manera:

```
0.3 => float mydur;
myDur :: second => now;
```

¡Existe una forma mejor! Ahora aprenderás sobre dos otros tipos de datos que están incluidos en ChucK, específicamente para controlar tiempos y duraciones.

1.4 Tiempo en ChucK: se trata de now

Hasta el momento has estado haciendo sonido conectando una onda sinusoidal u otro oscilador al dac, definiendo su frecuencia y ganancia y haciendo ChucKing de alguna duración, como 0.3 :: second, a una palabra clavemágica llamada now. Ahora profundizaremos en el mecanismo de tiempo de ChucK, introduciendo dos nuevos tipos de datos (como float e int) para lidiar con tiempo y duraciones.

¿Qué es una palabra clave?

> Las palabras clave son palabras reservadas espciales en un lenguaje de programación y sistemas. Ya hemos usado muchas palabras clave, como dac, SinOsc, y second. Puedes comprobar si una palabra es una palabra clave cuando la escribes o la ves en el miniAudicle, porque cambia de color de forma automática. Lo único que debes recordar como programador es que no puedes usar palabras clave para nombrar variables. ChucK no te lo permitirá, porque esas

> palabras ya han sido definidas (por los autores de ChucK) para usos especiales. Regla de oro: si miniAudicle le cambia el color a una palabra, no puedes usarla como nombre de una variable (excepto por palabras en los comentarios, que automáticamente son convertidas a color verde).

1.4.1 Variables de tipo dur

Ya has hecho ChucKing de duraciones de tiempo a la palabra clave de ChucK now para controlar tiempo en todos los ejemplos anteriores, pero puedes mejorar la canción "Twinkle" aún más usando un nuevo tipo de datos llamado dur, por duración.

Listado 1.11 Twinkle con variables tipo dur

```
/* Música Sinusoidal usando variables dur
pro un programador de ChucK
enero 2025

SinOsc s => dac;

//Bloque 1
//Define como variables las duraciones de las notas
220.0 => float twinkle;
0.55 :: second => dur onDur;
0.05 :: second => dur offDur;

1 => int onGain;
0 => int offGain;

//Toca una nota
twinkle => s.freq;
//Bloque 2
```

```
//Espera mientras el sonido suena
onGain => s.gain;
ondur => now;

//Bloque 3
//Luego espera durante el espacio ente notas
offGain => s.gain;
offDur => now;

//Bloque 4
//Frecuencia de la siguiente nota
1.5 *=> twinkle;

//Bloque 5
//Toca otra nota del segundo "twinkle"
twinkle => s.freq;
onGain => s.gain;
onDur => now;

offGain => s.gain;
offDur => now;
*/
```

En el bloque 1, haces dos variables de tipo dur, que inicializas a distintos valores. Una controla la cantidad de tiempo en que la onda sonora está sonando en el bloque 2, mientras que la otra controla el tiempo en que no suena en el bloque 3; este tiempo en silencio es a menudo llamado descanso entre los músicos. En el bloque 4, puedes usar aritmética de variable en el lugar para cambiar la frecuencia de twinkle para poder usarla con el segunda par de notas twinkle2; para músicos y físicos, este intervalo musica es llamado quinta perfecta. en el bloque 5 se prende y apaga nuevamente el oscilador, avanzando el tiempo usando tus variables tipo dur, para tocar la siguiente nota.

Entender tanto tiempo como duración (y variables tipo float e int) es esencial para programadores de ChucK. Por defecto, ChucK provee las palabras clave especiales de duración mostradas en la tabla 1.1. Probablemente sabes qué significan minute, hour, day y week, pero los valores en las tabla que podrías no conocer son ms y samp. Una milésima de segundo es llamada milisegundo, y su abreviación es ms. Entonces 1000 :: ms es equivalente a 1 :: second.

La duración de samp depende de la tasa de muestreo a la que tu hardware de sonido está corriendo, pero es siempre la unidad fundamental de tiempo en cualquier sistema de audio digital. Un samp es usualmente 1/44100 de segundo (porque se usan 44100 muestras de audio para capturar un segundo de audio). Por razones que son tanto técnicas (sincronización de audio a los formatos de video más comunes) como perceptuales (una tasa de muestreo que sea suficientemente alta para capturar el rango audible de escucha humana), 44100 es la tasa de muestreo más común usada en CDs, MP3s y otros dispositivos de almacenamiento.

Tabla 1.1 Palabras clave especiales de ChucK para duración y su correspondiente duración

Palabra clave	**Duración**
samp	1 muestra digital en tiempo ChucK (usualmente 1/44100 de segundo)
ms	1 milisegundo
second	1 segundo
minute	1 minuto

hour	1 hora
day	1 día
week	1 semana

PROBLEMA

> ¿Por qué los diseñadores de ChucK no definieron una palabra clave "year" (año) o "month" (mes)? Respuesta: mes y año no tienen una duración específica. Los meses pueden variar entre 28 y 31 días, mientras que los años pueden tener 365 o 366 días. ChucK es muy preciso con el tiempo, por lo que mes y año no fueron incluidos.

1.4.2 La importancia del tiempo

Ahora que has usado un poco tanto time como dur, hablemos un poco más en profundidad sobre por qué el tiempo es tan importante en ChucK y por qué es tan importante para ti como programador que entiendas cómo ChucK maneja el tiempo. El sonido es un medio basado en el tiempo. El sonido, y por extensión, la música, ocurren a través del tiempo. Sin el paso del tiempo, no habría sonido. Esta relación fundamental entre tiempo y sonido está en el núcleo del lenguaje ChucK, y permite el gobierno de cómo el tiempo fluye para así lograr que el sonido ocurra - y para controlar el sonido de forma precisa a lo largo del tiempo.

Existen unas pocas cosas en ChucK que funcionan en conjunto para hacer esto posible:

- time y dur son tipos de datos nativos, tal como lo son int y float.
- La palabra clave now contiene el tiempo de ChucK actual, que empieza en cero cuando haces click en el botón Start Virtual Machine

del miniAudicle.
- Por medio de la manipulación de now (haciendo ChucKing de una duración a ella), y solo por la manipulación de now, puedes causar el paso del tiempo en un programa de ChucK para generar sonido.

En ChucK, el tiempo (time) y la duración (dur) son tipos de datos básicos con los que puedes trabajar tal como números enteros y de punto flotante. Ellos representan valores de un tipo particular. Puedes declarar variables de estos tipos y definir sus valores y hacer aritmética sobre ellas.

1.4.3 Variables de tipo time

El tipo time contiene un punto en el tiempo (referenciado desde cero cuando la máquina virtual de ChucK es iniciada). El tipo dur almacena una duración, lo que es una cantidad de tiempo, el espacio entre dos momentos. Tal como podrías coordinar una juntarte con alguien en un momento en particular - "Te veré a las 1:00 p.m." - o especificar algo con una duración - "Estaré de vuelta en una hora" - es la naturaleza de time y dur en ChucK. Una variable time especifica un punto en el tiempo (la duración desde que ChucK empezó) y dur especifica una cantidad de tiempo. Por ejemplo, 2 :: second en ChucK es un valor de tipo dur. Puedes sumar duraciones para obtener nuevas duraciones (por ejemplo, 2 :: second + 1 :: minute) y almacenar duracioens en variables recién declardos, por ejemplo:

```
//duración de nota negra
0.8 :: second => dur quarter;
```

La duración de una nota negra en música depende del tempo, a menudo

expresado en términos de cuántas notas negras ocurren por segundo, también llamado BPM (beats per minute). Aquí hemos definido la duración de una nota negra como 0.8 segundos, lo que significa que nuestro tempo es de 75 BPM (60 segundos por minuto / 0.8 segundos por nota negra).

Las variables pueden ser reusadas para crear nuevas variables, por ejemplo:

```
4 :: quarter => dur whole;
```

Una vez que has definido la duración de una nota negra (y por lo tanto el tempo), puedes definir la duración de una nota redonda, que equivale a cuatro negras. De forma similar puedes definir la duración de las corcheas (negra / 2) y blancas (2 * negra o redonda / 2), así:

```
4 :: quarter => dur whole;

4 * quarter => dur whole;

whole / 2 => dur half;

quarter /2 => dur eighth;
```

La palabra clave especial now está en el corazón del trabajo con tiempo en ChucK, y es del tipo time. La palabra now tiene dos funciones especiales. Primero, cuando es leída, now contiene el tiempo lógico actual de ChucK. Esencialmente now es el reloj maestro de ChucK. Segundo, aunque now es una variable, no puedes cambiar su valor directamente, al menos no de la forma normal. Cuando tratas de cambiar el valor de now, por ejemplo, haciendo ChucKing de una duración particular a now, esto tiene el

importante efecto secundario de hacer que el tiempo fluya (y hacer que se genere sonido) durante precisamente esa duración de tiempo. En efecto, ¡ChucK está avanzando el tiempo hasta que now se convierte en el valor que quieres que sea!

La figura 1.8 muestra esto para el ejemplo "Twinkle" de dos notas del listado 1.11, ilustrando cómo el tiempo avanza entre bloques de código siendo ejecutados.

Otra manera de ver el tiempo en ChucK es que tu código detiene su ejecución durante cualquier duración que hagas ChucKing a now, pero todo lo relacionado a síntesis que hayas conectado (SinOsc => dac, por ejemplo) se mantiene andando y haciendo sonido. Cuando esa duración de tiempo en particular con la que hiciste ChucKing a now ha transucrrido, se ejecuta la siguiente línea de código.

1.4.4 Trabajando con now

Trabajar con now es simple y divertid pero es absolutamente esencial en la programación en ChucK - tienes que usarlo para hacer sonido, punto. Por la manera en que ChucK maneja el tiempo es tan importante y es lo que lo hace tan distinto de cualquier otro lenguaje, así que debemos reforzar algunos puntos sobre time, dur y el manejo de now. Aquí hay algunas cosas importantes a mantener en cuenta cuando trabajes con now.

- Hacer ChucKing de una duración a now hace que el tiempo de ChucK avance precisamente en esa cantidad: mientras el tiempo está avanzando tu código es automática y temporalmente suspendido (la siguiente línea no es ejecutada) y el sonido es generado por el sistema. Cómo lo hace depende de cómo hayas configurado la síntesis de

sonido.

- now nunca avanzará a menos que lo manipules. Entonces hasta que tú explícitamente hagas avanzar el tiempo, estás realmente trabajando dentro de un mismo instante en el tiempo.
- Otra manera de pensar sobre todo esto es que el código en ejecución de ChucK espera hasta que now se convierte en el tiempo que quieres alcanzar. Por esta razón, nunca deberás hacer ChucKing de una duración negativa a now, por ejemplo, -1 :: second => now. ChucK no puede (aún) viajar hacia atrás en el tiempo, por lo que tratar de lograr esto hará que tu programa se detenga.
- No existen restricciones en cuánto tiempo puede ser avanzado, mientras no sea una cantidad negativa. Entonces es posible avanzar el tiempo, digamos, un microsegundo, un samp, 2 horas o 51 semanas con el mismo mecanismo - depende ti; el sistema se comportará acordemente y de forma predecible.

Existen otras maneras de avanzar el tiempo; por ejemplo, haciendo ChucKing de un Event a now (como hacer click con el ratón del computador, presionando un botón de una palanca de mando, o tocando una nota musical en un teclado MIDI conectado a tu computador) para cuando no sabes de antemano cuánto tiempo avanzar. Dejaremos esto para una discusión posterior. Por ahora, has aprendido uno de los aspectos más importantes de ChucK: el control del tiempo usando duraciones.

Ahora que conoces los tipos de datos int, float, time y dur, deberíamos mencionar que ChucK incluye otro tipos de datos (llamados primitivos), incluyendo a string (una secuencia de caracteres como "hello world") y void (vacío, un tipo sin tipo, en caso que necesites una variable pero no necesites un tipo). Sabemos que esta noción de un tipo de datos void

puede ser confuso, pero lo necesitarás más adelante en el libro. La tabla 1.2 resume todos los tipos de datos incluidos en ChucK.

Tabla 1.2 Tipos de datos de ChucK

Tipo de dato	Descripción	Ejemplo	Comentario
int	entero	3, 3541	sin punto decimal
float	flotante	2.23, 3.14159, 22.0	con punto decimal
string	descripción	"hello" "data/a.wav"	comentario o texto
dur	distancia entre tiempos	1 :: second, ms, 3*day	duración
time	tiempo en ChucK	22050.0	tiempo en samples
void	sin tipo		

Has aprendido mucho de variables y datos y el uso de ellos para controlar parámetros de sonido y avanzar el tiempo. No obstante aún siguesescribiendo muchas líneas de códig para lograr cada nota. Tiene que haber una mejor manera, y sí, existe. Prosigamos.

1.5 Lógica y estructuras de control para tus composiciones

Para motivar las ideas de la siguiente sección, estudiemos un ejemplo distinto, no "Twinkle", que juegue con la modificación de altura y duración

(avanzando el tiempo) de una forma nueva y flexible. Las pocas líneas de código del siguiente listado pueden hacer sonidos muy interesantes, para siempre, moviéndose repetidamente a través del tiempo con now. Escribe este programa y ejecútalo. La magia está en que estás escuchando a ChucK tocar un flujo constante de notas, cambiando de manera aleatoria la frecuencia y duración de cada una.

Listado 1.12 Música aleatoria de onda triangular

```
/* Música aleatoria de onda triangular
por un programador de ChucK
*/

//Uso de una onda triangular para variar de los ejemplos anter
TriOsc t => dac;

//Bloque 1
//Bucle infinito que corre para siempre
while (true)
{
  //Bloque 2
  //Genera un número aleatorio entre 30 y 1000 para definir al
  Math.random2(30, 1000) => t.freq;

  //Bloque 3
  //Avanza el tiempo haciendo ChucKing a now de un número alea
  Math.random2f(30, 1000) :: ms => now;
}
```

Ahora juega con los números (30, 1000) del bloque 2. Cada vez que hagas un cambio, presiona el botón Replace Shred en el miniAudicle, para que el código que estabas ejecutando sea reemplazado con el nuevo código. Si

has hecho cambios significativos, ¡inmediatamente escucharás la diferencia! Esto es otro aspecto extremadamente potente de ChucK - la habilidad de modificar tu código sobre la marcha mientras el sonido todavía está siendo sintetizado.

Prueba esto

> No hagas click enel botón Replace Shred cada vez que le hagas un cambio al código del listado 1.12. En vez de eso, haz click en Add Shred. Ahora escucharás más magia de ChucK, que logra corriendo múltiples programas, llamados shreds, ¡al mismo tiempo! Puedes añadir más programas aleatorios sinusoidales, casi tantos como quieras, y escuchar que ChucK está felizmente generando muchos sonidos al mismo tiempo. Usarás este poder de ChucK y aprenderás pronto mucho sobre esto, pero por ahora, ¿no es acaso asombroso cuánto sonido/música puedes hacer con tan solo un poco de código? Hemos prometido que el cuociente entre sonido y escribir aumentaría.

Una vez que te hayas cansado de escuchar esto y estés listo para seguir, haz click en el botón Clear Virtual Machine (señalado con una gran X en la esquina superior derecha de tu ventana principal, como se muestra en la figura 1.9), y esto hará que el sonido se detenga.

¡Vaya! Con tan solo cuatro líneas reales de código ChucK del listado 1.12 has hecho más notas que las que podrías teclear. Vas a aprender un poco sobre lo que es posible a continuación, y aprenderás el resto en el capítulo 2.

El programa del listado 1.12 cambia aleatoriamente la frecuencia de una onda triangular en el bloque . Cada frecuencia nueva es aleatoria, y el tiempo que transcurre hasta que la siguiente es tocada es también

aleatoria. La línea while (true) del bloque 1 empieza lo que los programadores llaman un bucle infinito, lo que significa esencialmente "ejecuta para siempre todo lo que está dentro de las llaves". Mostraremos pronto más maneras de hacer bucles. En la última línea en el bloque 3 del cuerpo del bucle while, estás avanzando now en una cantidad aleatoria, punto en el que ChucK sabe que tiene que automáticamente suspender el código, dejar que el tiempo pase según la duración, y mientras tanto generar sonido. Precisamente después de que ese tiempo ha pasado, ChucK retoma la ejecución de tu código. Haciendo click en el botón Clear Virtual Machine le pone fin al bucle infinito y detiene el sonido.

Has escrito ahora tu primer bucle usando la palabra clave while. La palabra clave while en combinación con las llaves, se llama una estructura de contorl. A continuación revisaremos la forma exacta en que este bucle y otros funcionan. Aprenderás sobre lógica y estructuras de control, que son esenciales para hacer composiciones expresivas e interesantes a partir de tu código en ChucK.

1.5.1 El poder de la programación con declaraciones lógicas: if

Las declaraciones lógicas son verdaderas (true) o falsas (false) del tipo "siete no es iguala dos" o "-3 es un número negativo, y los números negativos son menores que cero". Las estructuras de control usan lógic para determinar cómo el código se ejecuta y cuáles son los efectos que tendrá. ChucK usa muchas estructuras de control standard presentes también en otros lenguajes de programación, incluyendo if (si), else (en otro caso), for (para cada) y while (mientras).

Empezaremos por observar detenidamente la declaración if. Todos usamos declaraciones if en la vida cotidiana, como "si tengo suficiente hambre, haré una parada y conseguiré comida". Las declaraciones if en ChucK pueden ser usadas para tomar decisiones en el código, basado en condiciones de variables u otros que almacenen valores. La figura 1.10 muestra un diagrama de lo que pasa con una condición if, avanzando procedimentalmente (línea por línea) a lo largo del códig (mostrado en el listado 1.13).

Listado 1.13 ejemplo de declaración if

```
//Cadena de sonido
SinOsc s => dac;

//Definir la frecuencia
220.0 => s.freq;
//Definir el volumen
0.6 => s.gain;

//Bloque 1
//Define un entero llamado chance para usar como variable lógi
//Su valor será 1 o 0
1 => int chance;

//Bloque 2
//Declaración if
//Si el valor dentro del paréntesis es igual a 1
//Ejecuta lo que está dentro de las llaves
//Si no lo es, omítelo
if (chance == 1)
{
  //Bloque 3
  //El sonido se reproduce solo si chance == 1
  1 :: second => now;
```

```
}

//Definir una nueva frecuencia para una nota distinta
330.0 => s.freq;
//Toca la siguiente nota
1 :: second => now;
```

Traduciendo el diagrama de la figura 1.10 a código, verás en el listado un programa simple que usa variable entera llamada chance en el bloque 1, que es inicializada a 1. Si chance es igual a 1 (chance == 1), entonces algo sucederá.

Con las estructuras de control que estás aprendiendo en esta sección, están usando los caracteres { y }. Puedes considerarlos como si fueran párrafos, pero los programadores los llaman bloques. En este caso, si la condición es true (verdad), entonces entras y ejecutas el código del bloque 3. Si la condición no es true, entonces sigues. También observa que se usan dos signos igual para expresar la prueba del condicional. Esta es la convención in ChucK, y en muchos otros lenguajes de programación, usar el símbolo == para indicar que estás haciendo una prueba lógica de igualdad, en este caso "¿es la variable llamada chance igual a 1?"

Existen más condiciones lógicas además de igual a, como menor que, mayor que, no igual a. Para esto, ChucK provee otros símbolos (llamados operadores relacionales) para probar valores, como se resume en la tabla 1.3.

Tabla 1.3 Condicionales lógicos

Símbolo	Significado en palabras	Ejemplo de uso
==	es igual a	if (x == 0)

!=	no es igual a	if (x != 0)
<	es menor que	if (x < y)
>	es mayor que	if (x > y)
>=	es mayor o igual a	if (x >= y)
<=	es menor o igual a	if (x <= y)

En el programa de frecuencia aleatoria de una onda triangular del listado 1.12, usas una palabra reservada, true, que siempre tiene l valor 1. En ChucK, el valor 1 es usado para representar true y el valor 0 para representar false. Si escribes y ejecutas esta línea de código ChucK

```
<<< true, false >>>;
```

verás que ChucK imprime

```
1 0
```

Observa que puedes imprimir dos ítems por medio de poner una coma entre ellos en tu declaración de imprimir <<< >>>. De forma similar, puedes escribir y ejecutar esto:

```
<<< -3 < 0, true == 1, true == false, 1 > 10 >>>
```

lo que resultará en

```
<<< 1, 1, 0, 0 >>>
```

lo que significa que -3 es menor que 0, que true es igual a 1, pero que true no es igual a false y que 1 no es mayor a 10. ¿Genial, no?

Continuando con el ejemplo del listado 1.13, como chance == 1, la prueba es true, por lo que el programa continua dentro del bloque y luego ejecuta la línea 3, avanza el tiempo 1 segundo, y luego toca por 1 segundo.

Ahora, ve y cambia la línea 1, configurando chance como 0, y volviendo a ejectuar el código recién editado. Esto en esencia cambia la línea 2 a un valor false, con lo que líena 3 nunca será ejecutada, por lo que no pasaré tiempo y no se emitirá ningún sonido durante el códig condicional. El programa continúa hasta tocar la siguiente nota en la línea 4, por lo que solo escuchas la nota alta. Entonces cuando chance == 1 escuchas dos notas y cuando chance == 0 escuchas solo una nota.

La declaración else va de la mano con la declaración if. Como puedes ver en la figura 1.11, la estructura if/else forma una bifurcación en el camino: si la condición es verdadera, anda a la derecha; si la condición es falsa, anda a la izquierda. Como puedes observar en el listado 1.14, la declaración else en el bloque 4 tiene su propio bloque, a continuación del bloque 3 de la declaración if. En este caso la prueba del bloque 2 será igual a false, cambiando la frecuencia a 330 Hz, avanzando el tiempo, y tocando el sonido por 3 segundos.

Si cambias la inicialización de chance a 1 en vez de 3 en el bloque 1, entonces la condición if del bloque 2 será verdadera y el primer bloque de código número 3 será ejecutad, tocando una onda sinusoidal de frecuencia 220 Hz por un segundp.

Listado 1.14 Ejemplo de código if/else

```
//cadena de sonido
SinOsc s => dac;

//define la frecuencia
220.0 => s.freq;
//define la ganancia
0.5 => s.gain;

//hloque 1
//variable lógica chance
3 => int chance;

//bloque 2
//si chance es igual a 1
if (chance == 1) {
  //bloque 3
  //ejecuta este bloque
  1 :: second => now;
}
//bloque 4
//en caso contrario, cambia la frecuencia de oscilación y toca
else {
  330.0 => s.freq;
  3 :: second => now;
}

//define y toca una nota, una octava arribe de twinkle
440.0 => s.freq;
1 :: second => now;
```

Corre el código del listado 1.14, cambiando el valor de chance. Escucharás la nota grave twinkle seguida de la nota mucho más alta solo cuando chance == 1, y para cualquier otro valor escucarás la nota twinkle2 seguida de la nota alta.

1.5.2 Operadores lógicos y condiciones

Tal como tienes expresiones lógicas que combinan o requieren múltiples condiciones ("Mi medidor de gasolina indica menos de la mitad del estanque, y estoy seguro que la gasolina está barata en este momento, así que compraré gasolina"), es también posible tener múltiples condiciones en una declaración if. El operador && (conocido como and) significa que todas las condiciones deben ser true para que la condición general sea true. El operador || (conocido como o) significa que basta con que una de las condiciones sea true para que se ejecute el bloque de código asociado. Un ejemplo de or podría ser "Si mi estanque está realmente vacío o la gasolina está muy barata, entonces llena el estanque". Los ejemplos de código ChucK de esto se encuentran en el listado 1.15.

Las declaraciones if/else/and/or se convertirán en extremedamente útiles a medida que vas aprendiendo más sobre programación musical.

Listado 1.15 Condiciones lógicas más complejas

```
//Declaraciones condicionales

//Condición or, si chance es igual a 1 o 5, entonces la condic
if ( (chance == 1) || (chance == 5) ) {
  //aquí va el código
}

//Si chance es menor que 2 y chance2 es mayor o igual que 6, e
if ( (chance < 2) && (chance2 >= 6) ) {
  //aquí va el código
}
```

1.5.3 La estructura de control de bucle for

Puedes hacer música aún más interesante usando la siguiente estructura de control, llamada bucle for. Como el nombre bucle sugiere, esta estructura de control es usada para crear bucles y comportamientos cíclicos. En el listado 1.16 y también ilustrado en la figura 1.12, un bucle for empieza definiendo una condición inicial con una variable (1). Contiene una condición (2) que confirma si es true o no (como una declaración if). Si es true, entonces ejecuta un bloque (párrafo) de código (4), y al final ejecuta un comando de actualización (3). Aquí la variable inicial es actualizada a un nuevo valor, y luego la condición es comprobada nuevamente. El proceso continúa hasta que la condición es false y se acaba el bucle.

Existen muchos casos musicales donde podrías querer usar un bucle for, como en el caso de tocar ascendentemente todas las notas de una escala (siguiente capítulo) o barrer gradualmente desde una frecuencia hasta otra, incluyendo todas las frecuencias entre medio. Aquí vemos un ejemplo de contar ascendentemente e imprimir enteros.

Listado 1.16 El bucle for

```
//Bucle for

//Comienzo de la estructura for
//(1) Configuración inicial, usualmente declara e inicializa u
//(2) Test condicional
//(3) Actualización  a hacerse cada vez que se ejecuta el bucl
for (0 => int i; i < 4; i++)
//Bloque de código a ejecutarse cada vez, mientras el condicio
{
  //Imprimir el valor de i
  <<< i >>>;
```

```
    //Avanzar el tiempo
    second => now;
}
```

En el ejemplo del listado anterior, la variable entera i es inicializa a 0 (1). El test condicional (2) es por supuesto true (0 es menor que 4), así que el bloque de código (4) es ejecutado (imprimir i, esperar 1 segundo); luego i es incrementado (3) (i++ es equivalente a 1 +=> i, que es lo mismo que i + 1 => i) y el bucle for es ejecutado de nuevo con i igual a 1. El test condicional sigue siendo true (1 es menor que 4), así que el bloque es ejecutado, i es incrementado y el bucle for es ejecutado de nuevo. Esto continúa hasta que i es igual a 4, y el test condicional ahora es false, por lo que el bucle termina.

Añadamos sonido a este concepto para que puedas empezar a escuchar lo que un bucle for puede hacer musicalmente. En programación musical, los bucles for pueden ser usados para tocar escalas, repetir patrones como líneas de bajos o melodías, cambiar ganancias de forma suave, hacer barridos en frecuencia hacia arriba o abajo, y muchas otras cosas super-musicales.

En el listado 1.17, usarás el bucle for (1) para definir frecuencias de tu SinOsc (3) según valores entre 20 Hz a 400 Hz (e imprimirlos (2)) cada 10 milisegundos (4). ¡Imagina tener que escribir todas estas líneas de código una a una como hiciste en la sección 1.2! La repetición en bluce te permite hacer sonidos y composiciones muchos más complejos y te ahorra muchas líneas de código. Desde ya, como compositor, eres capz de escribir música que tú nunca podrías haber escrito sin haber aprendido a programar.

Listado 1.17 Uso musical de un bucle for

```
//Onda sinusoidal conectada al dac
SinOsc s => dac;

//bucle for musical
//(1) bucle for entre 20 y 399
for (20 => int i; i < 400; i++) {
  //(2) imprime el valor actual de i
  <<< i >>>;
  //(3) define la frecuencia
  i => s.freq;
  //(4) espera 10ms y empieza de nuevo el bucle
  10 :: ms => now;
}
```

1.5.4 La estructura de control de bucle while

La última estructura de control que aprenderás en este capítulo es el bucle while (figura 1.13), que ya has usados para hacer un bucle infiinito. while es similar al bucle for pero se escribe de una manera distinta. Puedes reescribir el mismo programa de bucle for, usando un bucle while, como se muestra en el listado 1.18.

Listado 1.18 Uso musical de un bucle while

```
//Onda sinusoidal conectada al dac
SinOsc s => dac;

//(1) inicializa el contador con el número 20
20 => int i;
```

```
//(2)bucle while musical, tiene solo un test condicional
while (i < 400)
//(3) bloque de código a ejecutarse
{
  <<< i >>>;
  i => s.freq;
  10 :: ms => now;
  //(4) incrementa el contador (¡muy importante!)
  i++;
}
```

Después de configurar tu usual onda sinusoidal, inicializas el entero i en 20 (1). Luego defines la condición while (2) (while i sea menor que 400). El nombre sugiere que es una condición (while significa mientras), lo que puede hacerse un poco más obvio y fácil de recordad que el bucle for. El bucle whiel entonces continúa dentro de su bloque (3). Observa que tienes que agregar un comando de refresco (4) similar al de refresco del bucle for. Ejecutando este código, verás que hace el mismo sonido que el bucle for del listado 1.15. Es importante que aprendas que tanto los bucles for como los while porque los usarás de formas distintas a través de tu camino de aprendizaje de ChucK.

Observa que si te olvidas de incluir la línea de incremento (4) en el bloque de código del listado 1.18 el programa correría para siempre, con el valor de i y de la frecuencia de la onda sinusoidal siempre igual a 20. En la mayoría de los lenguajes de programación la noción de un bucle infinito (un bucle condicional que nunca cunmple con su condicion de término) debe ser evitado a toda costa. Pero esto no es tan así en ChucK, de hecho ya has usado un bucle infinito de forma intencional, while (true) en el listado 1.12, para hacer música bastante interesante. Los botones en la parte superior de miniAudicle para añadir, reemplazar y remover shreds y vaciar

la máquina virtual te permiten crear y usar bucles infinitos sin temer. Una característica atractiva de ChucK es que puede seguir corriendo mientras tú añades, modificas, reemplazas y agregas capas para hacer sonidos y música realmente interesantes.

1.6 Uso de múltiples osciladores en tu música

Hasta el momento en tus programas has tocado notas solas y melodías usando un oscilador. Como la música involucra más que melodías y ritmos, te podrás preguntar, ¿cómo puedo controlar y tocar múltiples osciladores al mismo tiempo? Esto es sencillo en ChucK, como se muestra en el siguiente listado.

Lsitado 1.19 Usando más de un oscilador

```
//Tu onda sinusoidal usual
SinOsc s => dac;
/(1) Un nuevo oscilador de onda sinusoidal
SinOsc s2 => dac;

//Definir las frecuencias y ganancias
//(2) Definir la frecuencia de la primera onda
220 => s.freq;
//(3) Definir la frecuencia de la segunda onda
1030 => s2.freq;
//Hacer que las ganancias sean 1/2.
0.5 => s.gain;
0.5 => s2.gain;

//Hacer que el tiempo transcurra para poder escuchar el sonido
```

```
second => now;
```

Un aspecto agradable de ChucK es que cuando conectas más de una fuente de sonido a algo, como el dac, los sonidos son automáticamente sumados. Los ingenieros de sonido le llaman a esto mezcla. En este ejemplo, cuando conectas el segundo SinOsc, s2, al dac (1), automáticamente será mezcoado con el otro SinOsc, y escucharás el sonido de ambos osciladores. Cofiguras las frecuencias para que sean distintas (2), (3), para que puedas escuchar que son dos osciladores distintos. Si defines que las frecuencias sean idénticas, escucharías un sonido a mayor volumen, porque mezclar dos señales idénticas genera una versión de mayor volumen de esas señales. En (4), haces que sus ganancias sean 0.5, porque quieres evitar saturar la salida de audio. Una buena regla de oro cuando mezcles un número de fuentes sonoras es escalar sus ganancias para que la suma de todos resulta en aproximadamente 1.0. Si añades otros oscilador SinOsc, s3, a esta mezcla, deberías hacer que las ganancias sean 0.3, o quizás una 0.5 y las otras dos 0.25.

1.7 Un ejemplo final: "Twinkle" con osciladores, variables, lógica y estructuras de control

Ahora es tiempo de reunir todo para hacer una composición usando todas las herramientas que has aprendido en este capítulo. El programa de ejemplo del listado 1.20 muestra cómo hacer justamente eso, aumentando tu ejemplo "Twinkle" para hacer algunas armonías y otras cosas interesantes con bucles. Primero, declararás dos osciladores, uno

sinusoidal (1) y otro triangular (2). Luego declararás variables para almacenar altura (3) (melodía) y tu nota con volumen (4). Finalmente en tu configuración inicial, definirás una variable de duración para almacenar la duración de las notas (5).

Listado 1.20a Armando "Twinkle", ¡con dos ondas!

```
//Twinkle, con dos osciladores

//(1) oscilador de onda sinusoidal
SinOsc s => dac;

//(2) oscilador de onda triangular
TriOsc t => dac;

//(3) define la altura inicial
110.0 => float melody;

//(4)ganancia para nota encendida
//ganancia para nuestra melodia con la onda triangular
0.3 => float onGain;

//(5)duraciones de notas
//usaremos esto para los tiempos de encendido y apagado
0.3 :: second => dur myDur;
```

Continuando con el ejemplo de "Twinkle" con dos osciladores (listado 1.20b), primero tocarás solamente con el oscilador de onda triangular (4), definiendo la ganancia del oscilador a 0.0 (7). Partirás la canción haciendo un barrido de altura hacia arriba con la onda triangular, yendo de 110.0 a 120.0 usando un bucle while (8). Luego aumentarás la frecuencia en 1 Hz cada vez que se ejecute el bucle (9) y lo harás rápidamente, actualizando

cada 1/100 de segundo (10).

Listado 1.20b Armando "Twinkle", todas las partes (parte B. barrido ascendente)

```
//solo toca t al principio, barrido ascendente de altura

//(6) enciende el oscilador de onda triangular
onGain => t.gain;

//(7) enciende el oscilador de onda sinusoidal
0 => s.gain;

//(8) itera hasta que la altura llegue a 220
while (melody < 220.0) {
  melody => t.freq;
  //(9) aumenta la altura en 1 Hz
  1.0 +=> melody;
  //(10) cada 1/100 de segundo
  0.01 :: second => now;
}
```

Una vez que hayas barrido ascendemente la altura, empezarás a tocar la melodía y la armonía (listado 1.20c) prendiendo el oscilador sinusoidal (10), definiendo su frecuencia a 110.0 (12), y tocando dos notas en el oscilador de onda triangular (definir gain como una valor distinto de ero) (14), esperar un momento (15), y luego apagarlo (16) y esperar de nuevo (17). Esto toca la primera parte de la canción "Twinkle".

Listado 1.20c Armando "Twinkle", primer Twinkle

```
//enciende ambos osciladores, define las alturas
```

```
//(11) ahora enciendo también el oscilador sinusoidal
0.7 => s.gain;
//(12) inicializa su altura
110 => s.freq;

//(13) usa un bucle for para tocar dos notas
for (0 => int i; i < 2; i++) {
  //(14) enciende la onda triangular
  onGain => t.gain;
  //(15) deja que la nota suene
  myDur => now;
  //(16) apaga la onda triangular
  0 => t.gain;
  //(17) silencio para separar las notas
  mydur => now;
}
```

Para el segundo twinkle, define nuevas alturas (18) y toca dos notas adicionales (19) de la misma forma, como se muestra a continuación.

Listado 1.20d Armando "Twinkle", segundo Twinkle

```
//(18) define la nueva frecuencia para twinkle
138.6 => s.freq;
1.5 * melody => t.freq;

//(19) toca dos veces con el bucle for
//dos notas adicionales, segundo twinkle
for (0 => int i; i < 2 ; i ++) {
  onGain => t.gain;
  myDur => now;
  0 => t.gain;
  mydur => now;
}
```

Como mostramos en el siguiente listado, para tocar "little", debes definir más alturas (20) y tocar dos notas más (21), de nuevo de la misma manera. Para "star" necesitar tocar solo una vez, así que define las alturas (22) y toca las notas (23), esta vez por 1 segundo (24).

Listado 1.20e Armando "Twinkle", tocando "little" y "star"

```
//(20) define la nueva frecuencia para "little"
146.8 => s.freq;
1.6837 * melody => t.freq;

//(21) toca dos veces con el bucle for
for (0 => int i; i < 2; i++) {
  onGain => t.gain;
  myDur => now;
  0 => t.gain;
  mydur => now;
}

//(22) define la nueva frecuencia para "star"
138.6 => s.freq;
1.5 * melody => t.freq;

//(23) toca esa nota
onGain => t.gain;
//(24) durante un segundo
second => now;
```

Para concluir la canción, mostrado a continuación, usa un bucle for (25) para barrer las frecuencias de ambos osciladores en forma descendente hasta ser cero. Una vez más, actualiza las frecuencias de los osciladores solo un poco (una caída de 1 Hz por cada pasada del bucle) pero muy seguido (cada 1/100 de segundo (26)).

Listado 1.20f Armando "Twinkle"

```
//(25) usa un bucle for para barrer de forma descendente desde
for (330 => int i; i > 0; i--) {

  i => t.freq;
  i*1.333 => s.freq;

  //(26) refresca cada 1/100 de segundo
  0.01 :: second => now;
}
```

Nota que en este ejemplo usaste casi todo lo que hemos aprendido en este capítulo. Usaste ondas sinusoidales y triangulares, variables int y flaot, bucles for y while, tests condicionales, matemática, y la palabra clave now con duraciones para controlar el tiempo. Introducimos un nuevo aspecto del bucle for aquí: puedes contar hacia abajo (desde un número mayor a un número menor) en un bucle for. En este caso, barres tu oscilador en forma descendente por medio de la inicialización de la variable entera i y luego con una cuenta hasta cero usando el operador de decremento (i--, que es lo mismo que 1 -=> i, que es lo mismo que i - 1 => i).

Ejercicio

> Trata de cambiar el número de veces que las notas son tocadas por medio de cambiar las condiciones de los bucles (prueba con i < 3 o 4 o más). Prueba cambiando el incremento (9) (listado 1.20b) y el decremento (25) (listado 1.20f) para tus bucles de barrido de altura. ¡Experimenta!

1.8 Resumen

¡Uf! Hemos cubierto mucho hasta este punto:

- Has aprendido un poco sobre ondas de sonido en general, que son fluctuaciones viajando a través del aire.
- Hemos discutido ondas sinusoidales y osciladores y los tipos de osciladores incluidos en ChucK (SinOsc, SqrOsc, SawOsc y TriOsc).
- Has aprendido de variables y tipos de datos: int, float, dur y time.
- Hemos mostrado cómo manipular tiempo en ChucK. No se emite sonido a menos que tú, el programador, haga que el tiempo avance (por medio de hacer ChucKing de duraciones a now).
- Has estudiado condiciones lógicas y palabras reservadas como if, else, &&, true y false.
- Has experimentado con estructuras de bucle, for y while, que te permiten controlar tus programas y sonido/música sin tener que escribir cada evento de forma literal.

¡Felicítate a ti mismo, porque ahora eres un programador y un artista digital! No obstante, ChucK ofrece mucho más poder que esto y te invita a aprender más sobre programación y sonido, acústica, interacción humano-computador, entre otros. Por ejemplo, ¿cómo es que llegamos a esos números para las alturas que usamos en este capítulo? ¡ChucK te puede ayudar con eso! Y aprenderás inmediatamente en el siguiente capítulo.

Así que continua aprendiendo más sobre las bibliotecas includias en ChucK (funciones útiles que puedes usar) y arreglos (formas de organizar y manipular tus datos artísticos/musicales).