Programming for Musicians and Digital Artists

by Ajay Kapur, Perry Cook, Spencer Salazar, Ge Wang

Chapter 3

brief contents

3 Arrays: arranging and accessing your compositional data

This chapter covers

- Declaring and initializing arrays
- Retrieving and modifying data in arrays
- Storing different types of data in arrays
- Using arrays to control musical parameters for a song

Now that you've been introduced to the power of the Standard library and Math utilities, you're going to learn one more important thing in order to make the coding of your melodies and compositions much easier and more expressive and to allow you to create longer songs without excessive typing. This chapter covers arrays, which are super-powerful mechanisms that allow you to make collections of data that you can use for all sorts of things. Arrays are fundamental to how almost all computer programs work. They're how email, pictures, sound/music files, and everything else get stored and accessed inside computers, smartphones, tablets, and on the web.

An *array* is a collection of data stored in memory. For composer-programmers, one use for arrays is to store lists of musical parameters that you want to use to control your songs over time, such as musical pitches (or MIDI note numbers), volumes, times, and/or durations. Another use for arrays is to store lists of character strings such as song lyrics. Arrays can store any type of data, and you can even make arrays of unit generators like `SinOsc` or `Noise`, which you'll do in the next chapter.

So far you've generated melodies by using loops to increase or decrease frequencies or pitches, or you've made long programs that specify each note you want, line by line. Wouldn't it be great if you could make a list of the notes you want to play at the beginning of the program? What if you could somehow store your compositional parameters in memory, making it easier to write whole songs? That's exactly what arrays allow you to do.

3.1 Declaring and storing data in arrays

Here you'll begin to learn how to play the notes of our "Twinkle" song by storing all of them in a single array. In this case, you'll store your melody as a list of MIDI note numbers. To start out, let's visualize what an array is conceptually.

In figure 3.1 you see seven boxes containing integers. These boxes are seven pieces of memory attached to your array. Inside the boxes are integer numbers; in this case they represent the MIDI notes of our "Twinkle" melody. Beneath each box, you see numbers that are the indices to the boxes. Notice that the index starts at 0. All arrays start at index 0. The array at index 3 will give you the data in that box, in this case MIDI note number 64, the fourth note of "Twinkle."

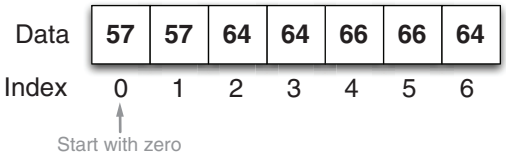

Figure 3.1 An array stores data in individual locations, accessed by an index number.

Now let's see how to represent this concept in code. You represent an array with square brackets (`[]`), with the number in the square brackets being the number of elements in the array. In this case there are seven MIDI notes, so you declare array `a` as `int a[7]` ❶ in the following listing. Notice that the array is of type integer (you can make arrays of any type, as you'll see later on). Next, you have to store your melody into the array. You do this by putting each integer into its own location in the array, one at a time, as shown in the lines between ❷ and ❸.

Listing 3.1 Declaring and filling an array of integers the long way

```
// array declaration (method 1)
int a[7];

57 => a[0];
```

❶ Declares an array of specific length (7).

❷ Sets value stored in the zeroeth element…

```
57 => a[1];
64 => a[2];
64 => a[3];
66 => a[4];
66 => a[5];
64 => a[6];

<<< a[0], a[1], a[2], a[3], a[4], a[5], a[6] >>>;
```

…and all locations…

❸ …up to and including the last element.

You now have your MIDI notes in the array, but this took way too many lines of code. Isn't there a shortcut? Of course there is! As shown in listing 3.2, you can store your MIDI notes all on one line of code. Notice that you use a new operator, `@=>`, which is a special ChucK operator used to store all of the data into your array `a`, all at once. `@=>` is called the at-ChucK operator or the explicit assignment operator. You can use this operator to copy other objects in ChucK, but by far the most common use is for copying lists of elements into arrays.

Listing 3.2 Declare and initialize an array all at once

```
[57, 57, 64, 64, 66, 66, 64] @=> int a[];

<<< a[0], a[1], a[2], a[3], a[4], a[5], a[6] >>>;
```

The other part of this new way of declaring and initializing an array is that now your declaration of array `a` doesn't have the preassigned `[7]` as the number of elements. This is part of what the `@=>` operator allows you to do. The `a` points to the beginning of your list of MIDI notes, but the size needn't be determined during declaration. ChucK figures this out for you, making it easy to add and subtract notes to make the melody longer or shorter by typing extra numbers into the list, without having to count how many notes you have each time.

3.2 Reading and modifying array data

Now you know a couple of ways to put data into an array: one element at a time or by copying a list through `@=>`, the at-ChucK operator. But how do you retrieve this data? Listing 3.3 shows how this is done. Here you read one element of the array in line ❶ and print the result in ❷. But what does this print? Think about it. Did you say 57? Remember that the index starts at 0! So the answer is really 64 (the third item in the array).

> **STARTING WITH ZERO** You might have been wondering why this book began with chapter 0 rather than chapter 1. That's because we're computer scientists, and we wanted to get you thinking right off the bat about things like array indices starting with zero rather than one. Interesting fact: many computer science buildings at universities number the ground floor as floor zero.

What if you want to change your melody from within your program? No problem; an array contains variables, so you can change what's in the array at any time by setting a new integer into the array `a[2]` location (or any location) ❸.

Listing 3.3 Accessing (reading and writing) data in an array

```
// declare and initialize an array
[57, 57, 64, 64, 66, 66, 64] @=> int a[];

// array look up by index
a[2] => int myNote;

// print it out to check
<<< myNote >>>;

// want to change data?  no problem! (print too)
61 => a[2];
<<< myNote, a[2] >>>;
```

1 Looks up note in array by integer index

2 Prints it

3 Changes array element value at index

This code would print in the Console window

```
64 :(int)
64 61
```

This output reflects the fact that you changed the data in that one element of your array (to 61), but the data you previously read from there and stored into the `myNote` variable is unchanged (still 64). The `myNote` variable is stored separately from the array data, so you can change one without affecting the other at all.

You might be wondering by now, "What if I want to change the size of the array after I've declared it?" It's possible to do that, as described in detail in appendix B, section B.6. But that's a bit advanced for the task at hand, so we'll move on.

3.3 Using array data to play a melody

You know how to make and fill arrays, so now you can finally use your array to control sound, playing the melody you've stored. You start out in listing 3.4 by declaring and connecting a `SqrOsc` as the source of your audio signal chain 1. Then you set up some gains to turn your notes on and off (0.7 for on and 0.0 for off) 2. Next you declare and initialize an array just as you did before, but this time it has more data (more notes of your song!) 3.

Now it's time for a new method: `a.cap()` (short for capacity), which tells you the total size of array `a[]`. In this case the size is 14. You can then use that number in a `for` loop to cycle through every element in array `a[]`. You print the value of index `i` and then the contents of the array at that index `a[i]` 4. Notice that when running the code, it matches the array picture from figure 3.1.

Everything comes together musically when your MIDI note from array `a[i]` is converted to frequency with the `Std.mtof()` method 5. After that, it's the standard means of playing notes: set the gain to something non-zero 6, advance time 7, and turn the note off for a while 8 before going back to play the next note. This ends when `i` reaches 14 and there are no more notes to play.

Listing 3.4 Playing a melody stored in an array

```
//  Let's Twinkle with a square wave
SqrOsc s => dac;                                    // 1 Square wave oscillator for melody

// gains to separate our notes
0.7 => float onGain;                                // 2 Note on/off gains
0.0 => float offGain;

// declare and initialize array of MIDI notes
[57,57,64,64,66,66,64,62,62,61,61,59,59,57] @=> int a[];   // 3 Array of MIDI notes (int) for melody

// loop for length of array
for (0 => int i; i < a.cap(); i++)
{
    <<< i, a[i] >>>;                                // 4 Prints index and array note

// set frequency and gain to turn on our note
    Std.mtof(a[i]) => s.freq;                       // 5 Sets pitch for melody notes
    onGain => s.gain;                               // Note on 6
    0.3::second => now;                             // 7 Duration for note on

// turn off our note to separate from the next
    offGain => s.gain;                              // Note off 8
    0.2::second => now;
}
```

3.4 Storing other types of data in arrays

You might have noted that the version of "Twinkle" produced by listing 3.4 is nearly complete, except for some problems with timing. For example, the notes corresponding with "star" and "are" should be longer than the others. Next, you're going to learn that you can store pretty much any type of data in arrays, including `int`s, `float`s, `string`s, and even durations.

3.4.1 Using an array to store durations

The "Twinkle" melody requires some notes to be longer than others. Those who write and read music call the note durations associated with "little" *quarter notes* and the longer note durations ("star") *half notes*. These are shown in musical notation in figure 3.2.

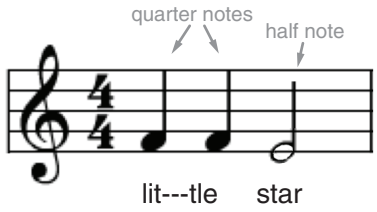

Figure 3.2 Quarter notes and a half note denote the different durations for our "Twinkle" song.

The reason quarter notes are called that is because there are four of them per musical *measure*; likewise, there are two half notes, twice as long, per measure.

To account for this in your code, you could change your program to use logic, so that it holds those two notes longer. If you replace line 7 in listing 3.4 with the `if`/`else` statement shown in listing 3.5, then the two notes will be held twice as long. The `noteOff` time of 0.2 seconds is the same in all cases. This time, added to a `noteOn` time

of 0.8 seconds ❶, give you half notes of one second. And the 0.3 second `noteOn` time plus a 0.2 second `noteOff` gives you the desired 0.5 second quarter notes ❷. Try it and you'll hear the difference!

Listing 3.5 New logic to control note durations

```
if (i==6 || i==13)
{
    0.8::second => now;          <-- ❶ Some notes are longer
}
else
{
    0.3::second => now;          <-- ❷ The rest are shorter
}
```

We've also said that arrays can store any type of data, which is really powerful for making music or any kind of programming in general. If you think of all the things you'd like to manipulate in order to make a proper song, you might come up with pitch, duration, loudness, and other things (even lyrics!). You could store any or all of those in arrays.

Let's store your note durations in an array, as shown in listing 3.6. Here you perform the very same setup as in listing 3.4. You first declare and connect the square wave oscillator ❶. Then you declare and initialize the gains for `noteOn` and `noteOff` ❷ and create your array of MIDI notes ❸ (now called `myNotes`, remembering what you learned about meaningful names for variables). You declare two variables of type duration, one for quarter notes called `q` ❹ and one for half notes called `h` ❺. You then use those shorthand variable names to declare and initialize an array of durations called `myDurs` ❻.

Once all that has been declared, you can drop into a `for` loop to play your song, accessing the note array the same way ❽, turning each note on by using the `onGain` variable you declared ❾. But now you can use your `myDurs` array to play each note for the correct duration by ChucKing the proper duration to `now` ❿. After time passes, you turn each note off ⓫ and ChucK your off-time duration to `now`. You repeat that for each note; then the loop ends when your counter variable `i` reaches the size of your note array `myNotes.cap()` ❼.

Listing 3.6 Storing durations in an array

```
//  Let's Twinkle with a square wave
SqrOsc s => dac;                 <-- ❶ Square wave oscillator for melody

// gains to separate our notes
0.7 => float onGain;             <-- ❷ Note on/off gains
0.0 => float offGain;

// declare and initialize array of MIDI notes
[57,57,64,64,66,66,64,62,62,61,61,59,59,57] @=> int myNotes[];
                                 <-- ❸ Array of MIDI notes (int) for melody
```

```
// quarter note and half note durations
0.3 :: second => dur q;
0.8 :: second => dur h;
[q, q, q, q, q, q, h, q, q, q, q, q, q, h] @=> dur myDurs[];

// loop for length of array
for (0 => int i; i < myNotes.cap(); i++)
{

    // set frequency and gain to turn on our note
    Std.mtof(myNotes[i]) => s.freq;
    onGain => s.gain;
    myDurs[i] => now;

    // turn off our note to separate from the next
    offGain => s.gain;
    0.2::second => now;
}
```

4 Duration for quarter notes

5 Duration for half notes

Array of durations for melody notes 6

For loop iterates over length of note array 7

8 Sets pitch for melody notes

Note on 9

For duration stored in array for that note 10

Note off 11

3.4.2 *Arrays of strings: text can be musical too*

Now that you're on a roll with storing things in arrays, let's put some words into an array of strings. Listing 3.7 shows just how to do this. As with your other arrays, you use the same square brackets [...] to hold your list of strings 1. You separate each element by commas and use the `@=>` copy form of the ChucK operator 2 to store that list into a newly declared array of type string, which you call `words[]` 3. Note here that each element can have a different length as an individual string ("Twin" versus "kle"), and ChucK just figures it all out for you, allocating just the right amount of storage to hold what you've declared.

> **Extending code across lines**
>
> Note here that you used a feature of ChucK coding wherein a single line of code can extend across multiple lines of text. The semicolon is what terminates an executable line of code in ChucK, so you can put half of your list of words, notes, and whatever on one line and the other half on the other (or even extend over many lines), and ChucK puts it all together when it finds the first semicolon.

Listing 3.7 An array of strings (the "Twinkle" lyrics)

```
// make an array to hold words and syllables
["Twin","kle","twin","kle","lit","tle","star,",
 "how", "I","won","der","what","you","are."] @=> string words[];
```

1 Declare and initialize array of strings for lyrics.

Object copy form of ChucK operator. 2

ChucK figures out how big to create words[] array. 3

3.5 Example: a song with melody, harmony, and lyrics!

To wrap up this chapter, we want to give you a more fleshed-out example of how to make a whole composition using all the things we've talked about so far. You'll make a much longer version of "Twinkle," using two MIDI note arrays for melody and harmony. You'll use a `float` array to hold the note durations. You'll also use an array of `string`s to print out the words as the song plays! And you'll pan the melody randomly, using the Math library. So let's get to it!

In listing 3.8, you first declare and add panning to your melody oscillator using the `Pan2` object ❶. Next, you make another oscillator for harmony and connect it to the `dac` ❷. As you've learned, the `dac` takes care of mixing these sources together automatically. Then you make some gain variables to use for turning your notes on and off ❸.

Next, you have the same melody array that you've been working with ❹, and you make another array to control another oscillator for harmony ❺. You make an array of floats that you'll use for durations ❻ and yet another array of type `string` that holds your words ❼.

Be careful to make all of these arrays the same length, or you might encounter an error when you ask for the `[i-th]` element if it doesn't exist because one of the arrays is shorter than the others.

You continue on to the same `for` loop as you had in the last example using the `.cap()` function to determine the size of the arrays you're using ❽. In the loop block, you print out the value of your counter index `i`, along with your melody/harmony MIDI note numbers and, most important, your word or word fragment corresponding with that particular note of the song ❾.

You then set the frequencies of your two oscillators ❿, but you add variety for the melody line, setting a random panning value for each note ⓫. You turn on your two oscillators by setting their gains to your previously defined `onGain` ⓬; then you advance time by reading and using the values in the `durs[]` array ⓭, ⓮.

> **NOTE** In listing 3.8 you use a shorthand method ⓬ of setting the values of multiple things to a common single value. `onGain => t.gain => s.gain;` works just fine, because first `onGain` is ChucKed to `t.gain`, which then has this new value, which is then ChucKed to `s.gain`.

Listing 3.8 "Twinkle" with melody, harmony, and lyrics!

```
// by ChucK Team, July 2050

// two oscillators, melody and harmony
SinOsc s => Pan2 mpan => dac;
TriOsc t => dac;

// we will use these to separate notes later
0.5 => float onGain;
0.0 => float offGain;
```

❶ SinOsc through Pan2 for melody

❷ TriOsc fixed at center for harmony

❸ Note on/off gains

```
// declare and initialize our arrays of MIDI note #s
[57, 57, 64, 64, 66, 66, 64,
62, 62, 61, 61, 59, 59, 57] @=> int melNotes[];
[61, 61, 57, 61, 62, 62, 61,
59, 56, 57, 52, 52, 68, 69] @=> int harmNotes[];

// quarter note and half note durations
0.5 :: second => dur q;
1.0 :: second => dur h;
[ q, q, q, q, q, q, h, q, q, q, q, q, q, h] @=> dur myDurs[];

// make one more array to hold the words
["Twin","kle","twin","kle","lit","tle","star,",
 "how", "I","won","der","what","you","are."] @=> string words[];

// loop over all the arrays
//       (make sure they're the same length!!)
for (0 => int i; i < melNotes.cap(); i++)
{
    // print out index, MIDI notes, and words from arrays
    <<< i, melNotes[i], harmNotes[i], words[i] >>>;

    // set melody and harmony from arrays
    Std.mtof(harmNotes[i]) => s.freq;
    Std.mtof(melNotes[i]) => t.freq;

    // melody has a random pan for each note
    Math.random2f(-1.0,1.0) => mpan.pan;

    // notes are on for 70% of duration from array
    onGain => s.gain => t.gain;
    0.7*myDurs[i] => now;

    // space between notes is 30% of array duration
    offGain => s.gain => t.gain;
    0.3*myDurs[i] => now;
}
```

4 Melody (int) MIDI note array

5 Harmony (int) MIDI note array

6 Duration (dur) array

7 Lyrics (string) array

8 Plays through all notes in array

9 Prints note data, including lyrics

10 Sets frequencies from array MIDI notes

11 Random pan for melody oscillator

12 Turns on both oscillators

13 70% of array duration is note on time

14 30% of array duration is off time

3.6 Summary

Arrays can make life much easier and more organized for you:

- Arrays can be used not only for storing integers, like melodies as note numbers, but also to store any sequence of floating-point numbers, such as gains or frequencies.
- Arrays can be used to store durations and strings, including words, directions, and even filenames, or any type of data—unit generators, pretty much anything!
- You can find the size of any array using the `.cap()` method, so you don't have to count elements each time you change something.

Now that you've learned quite a bit about the ChucK language, we can turn our attention to making sounds and music richer and more realistic. Our next chapter will dive into working with sound files (essentially arrays that hold sound) and how you can use and manipulate them to make your music even more awesome.