Arrays

Variables give us the ability to store individual numbers, strings, boolean values, or object addresses, but for many tasks we have to deal with dozens, hundreds, or thousands of such values. Instead of creating thousands of individual variables, we can instead use an *array* structure to hold a list of values that are all of the same type. Arrays have built-in numbering and we can use this to easily process values that live in an array using a for loop.

Array Structure

Suppose that we have a list of integer grades that we have to find the average for, and then print. If we don't know about arrays, the best we could do is create a variable for each grade, and name them grade0, grade1, etc. Then every task we want to do with this grades, we have to manually do line by line:

```
int grade0 = 67
int grade1 = 71
int grade2 = 99
// ... do grade3 through grade8
int grade9 = 86

double average = 0

average = average + grade0
average = average + grade1
// ... do grade2 through grade8
average = average + grade9
average = average / 10

print("Grades:")
print(grade0)
print(grade1)
// ... do grade2 through grade8
print(grade9)
```

Note that even in writing out this much, I was too lazy to write every line, I just said things like "do grade3 through grade8" to indicate that we want to do the same thing to the variable grade followed by a number that changes.

An array gives us a way to number variables in a list, where the number is a separate part that we can programmatically change in order to work our way through the list.

At the hardware level, to create an array we would set out a sequence of rows in Main Memory to hold our list of values – each row in the array acts like a separate variable for storing a value, but the array lets us treat the whole array as one object when we need to.

At the hardware level, to create an array we would set out a sequence of rows in Main Memory to hold our list of values. In many languages, the array itself is assigned the memory address of the first of these rows, and each subsequent value in the array will be on a new row[1]. So we can think of each position in the array as an offset from that first row. The first value is offset by 0, the next value is offset by 1, etc.

We use these numbers, indicating how many rows down to move in the array, as the array *index*, which is always an integer value (you can't move down half a row in memory). We will use square braces to surround the index, at the end of the name of the array. Since we start on the top row of the array, with no offset, indices[2] always start at 0.

The list of values stored in the array are its *elements*.

To create an array of any type, we have to say how many rows in memory to reserve for the whole list of elements. We also use square braces around this number when we create the array.

```
// declaring an array variable for an array
// of sizeOfArray elements
type[sizeOfArray] arrayVariableName;

// putting a value into an array at an index
arrayVariableName[index] = valueForElement;
```

Let's rewrite the average code above, using array syntax:

```
// declare an array of ints called grades
// and set it up to be size 10
     // “int array grades is size ten”
     // ”grade is an int array, size 10”
     // ”grade is an array of 10 ints”
int[10] grade
grade[0] = 67 // put 67 in 0th row of the grades array
grade[1] = 71 // put 71 on 1th row of the grades array
grade[2] = 99 // put 71 on 2th row of the grades array
```

[1] Actually, some types of values might take up more or less than a row in Main Memory, so each value in the list might be stored two rows down from the previous, or only half a row over if we are doing byte addressing for some reason. But even if it is only half a row, it is still in programming always represented as an integer unit, representing the size of the type.
[2] This is the plural of the word "index".

```
10  // ... do grade[3] through grade[8]
11  grade[9] = 86 // put 86 on 9th row of the grades array
12
13  double average = 0
14  // add the value from the 0th row of the array
15  average = average + grade[0]
16  // add the value from the 1th row of the array
17  average = average + grade[1]
18  // ... do grade[2] through grade[8]
19  average = average + grade[9]
20  average = average / 10
21
22  print("Grades:")
23  print(grade[0])
24  print(grade[1])
25  // ... do grade[2] through grade[8]
26  print(grade[9]);
```

Let’s look at what’s happening under the hood. In line 6 we are declaring the array variable and associating it with an array of 10 int elements. We will assume that array elements have default values, in this case, 0 for ints.

```
int[10] grade
```

	MM	*value*
Index: 0	100	0
1	101	0
2	102	0
3	103	0
4	104	0
5	105	0
6	106	0
7	107	0
8	108	0
9	109	0
	…	

Next we use the elements of the array as variables to store values (the code also assumes that we fill in the other rows, but let’s just look at what these lines would do:

```
grade[0] = 67 // put 67 in 0th row of the grades array
grade[1] = 71 // put 71 on 1th row of the grades array
grade[2] = 99 // put 71 on 2th row of the grades array
grade[9] = 86 // put 86 on 9th row of the grades array
```

	MM	value
Index: 0	100	67
1	101	71
2	102	99
3	103	0
4	104	0
5	105	0
6	106	0
7	107	0
8	108	0
9	109	86
	…	

All we have done so far is switch to array syntax, we haven't yet made use of the strength of arrays to simplify this code. We need a little more understanding first.

Array Index

Notice the somewhat odd wording I used, I called grade[0] the 0th row and grade[1] the 1th row. Talking about the "first" row of an array can be ambiguous: do I mean the very first one, which is at offset 0, or the first one as in the one numbered "1" which is actually the second row in the array, offset by 1? Because of this, people have come up with different ways to refer to the rows such as 0th and 1th [3] or always saying "index 0" and "index 1" or "offset 0" and "offset 1" etc.

When talking about code, we generally just say the name of the array followed by "sub" then the index so "grade sub 0" to mean grade[0] or "grade sub 1" to mean grade[1] or even leave out the sub and just say "grade 0" to mean grade[0] and "grade 1" to mean grade[1]

Because index always starts at 0, the last index in the array is always 1 less than the size we created the array at. We initialized grade with size 10, so grades runs from index 0 to index 9 – it takes up 10 rows, but those rows are numbered 0..9.

I was able to do math using the elements of the array and print the elements of the array, just like any other integers. At the moment variable names like grade[3] look strange because they are new, but they really are just variable names, and I can do anything with grade[3], an integer that lives in an array structure at index 3, that I could have done with an integer variable grade3, an integer that lives in an individual variable.

[3] Or aloud, pronouncing 1st as "wunst" to indicate index 1.

Using Values in an Array

So far, changing to array format hasn't bought us anything, we're still manually moving through just as many lines of code. This will change when we start using for loops with arrays. Let's look at a few more array examples first

```
// declare variable rents as an array that holds
// 10 double values, indices 0 to 99
double[100] rent
// put the value 1560 at index 0 in the rent array
rent[0] = 1560.00
// ... fill in rent[1] to rent[89]
// put the value 1290 at index 99 in the rent array
rent[99] = 1290.00

// using a variable to hold the index
double myAptNum = 67
rent[myAptNum] = 1350.00
// same as
rent[67] = 1350.00

// my rent is stored on row 22
// and I have to pay $132 in utilities
double monthTotal = rent[22] + 132.00

// declare variable petNames as an array
// that holds 15 string values
// indices 0 to 14
string[15] petNames
// put the value "Fluffy" at index 0 of the petNames array
petNames[0] = "Fluffy"
// . . .  fill in petNames[1] to petNames[13]
// put the value "Spot" at index 14 of the petNames array
petNames[14] = "Spot"

// my pet is on 4th row (index 3) and
// I am printing about a vet appointment
int x = 4
print(petNames[x - 1] + " has a vet appointment on May 3.")

// declare variable catCanFly as an array
// that holds 5 boolean values
// indices 0 to 4
boolean[5] catCanFly
```

```
// put the value true at index 0 of the catCanFly array
catCanFly[0] = true
// . . .  fill in catCanFly[1] to catCanFly[3]
// put the value false at index 4 of the catCanFly array
catCanFly[4] = false

// get which row their cat is on and then
// advise them on what activity to do
int catIndex = prompt("what index is your cat at in the array?")

if (catCanFly[catIndex]) {
    print("Go ahead and take your cat flying");
} else {
    print("Just play with your cat on the ground");
}
```

Since the array index is always an int, no matter what type is in the array, we can use literal int values, or anything at evaluates to an integer, like an int variable or an expression, in the square braces to indicate which row in the array we want, which we did in the above code.

CS Numbers vs Human Numbers

At the end of that code, we asked the user for the array index to use. Most of the time, users would be confused by using "computer scientist numbers" that start at 0, so we would adjust between these and "human numbers" that start at 1 when talking to a user:

```
// get which row their cat is on
int catIndex = prompt("which cat (1 to 5)?")

// they entered a human number
// subtract 1 to make it a correct array index
catIndex = catIndex - 1

// advise them on what activity to do
if (catCanFly[catIndex]) {
    print("Go ahead and take your cat flying");
} else {
    print("Just play with your cat on the ground");
}
```

Array Initialization

Like other variables, different languages have different ways of determining the values stored in an array when it is first declared. In some languages they must be initialized before being used, in some languages they have default values, such as 0 for numbers, and in some languages they contain whatever garbage values happened to be left on those rows in memory the last time they were used.

For pseudocode examples, we will assume that arrays start with the default values 0 for numbers, false for booleans, and null for everything else.

In many languages, if we know the values we want for an array when it is created, we can do a *special initialization* to put them all into place at once. We do this by putting them into a comma separated list inside curly braces. In this case we wouldn't specify the array size in the square braces, since it can be read by reading the number of elements in the curly braces:

```
// special initialization
string[] zooAnimals = {"elephant", "zebra", "hippo", "penguin",
     "aardvark"}

// same as saying
string[5] zooAnimals
zooAnimals[0] = "elephant"
zooAnimals[1] = "zebra"
zooAnimals[2] = "hippo"
zooAnimals[3] = "penguin"
zooAnimals[4] = "aardvark"
```

Array Size

Array structures are static, which means that whatever size they were when we created the array, they stay that size and cannot change. In some languages, you can't even use a variable when setting the size of an array, you can only create an array using a literal number (or constant) for the size!

Trying to put data outside the size of the array is a problem. No array has negative indices, and remember that valid indices in the array always go up to one less than the size.

Suppose we used an invalid index in an array. Depending on the language this could be a syntax error, or a runtime error that causes the program to halt, or this could simply access the row in main memory at that offset, even though it isn't actually part of the array... this would allow us to overwrite or read from whatever happens to be in the surrounding rows of memory that happen to be near the array! This is very dangerous and has been used in various types of malicious coding.

In some languages, arrays know their own size, while in others it is the programmer's responsibility to keep this value in another variable. For our pseudocode, we will assume that arrays know their size, and that we access it using the name of the array followed by `.size`.

```
// declare variable petNames as an array
// that holds 8 string values
// indices 0 to 7
// so petNames.size is 8
string[8] petNames
petNames[0] = "Muffy"
petNames[1] = "Fluffy"
petNames[2] = "Clyde"
print("there are " + petNames.size + " pet names in our list")

// put "Rover" in the last position in petNames
// which must be its size minus 1
petNames[petNames.size - 1] = "Rover"
```

For the purpose of examples, I will put the size at the end of the array when showing examples, so at the end of the previous code the array might look like:

variable	*MM*	*value*
Index: 0	100	"Muffy"
1	101	"Fluffy"
2	102	"Clyde"
3	103	null
4	104	null
5	105	null
6	106	null
7	107	"Rover"
length	108	8
	…	

For Loops for Arrays

The real power of arrays comes from the fact that we can use a variable in the square braces, and use a for loop to change that variable. As a result, we can write a for loop to do the same thing to each element of the array. We want to end up writing code like

```
// add up all the grades in the array
for (int i = 0; i < grade.size; i = i +1) {
    total = total + grade[i]
}
```

But let’s work our way up to that, to make sure we understand what we’re doing.

First, let’s revisit doing all the steps of an average manually, using array syntax:

```
// new list of grades using special initialization
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49}
double average = 0
double total = 0

// go through each grade in the array, adding it onto the total
total = total + grade[0]
total = total + grade[1]
total = total + grade[2]
total = total + grade[3]
total = total + grade[4]
total = total + grade[5]
total = total + grade[6]
total = total + grade[7]
total = total + grade[8]
total = total + grade[9]

// math for the average
average = total / 10
```

Since we can use an int variable in the square braces as the array index, we could rewrite the above as:

```
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49}
double average = 0
double total = 0

// variable to use as index
int i = 0
total = total + grade[i]
i = i + 1 // now the index is 1
total = total + grade[i]
i = i + 1 // 2
total = total + grade[i]
i = i + 1 // 3
total = total + grade[i]
i = i + 1 // 4
```

```
total = total + grade[i]
i = i + 1 // 5
total = total + grade[i]
i = i + 1 // 6
total = total + grade[i]
i = i + 1 // 7
total = total + grade[i]
i = i + 1 // 8
total = total + grade[i]
i = i + 1 // 9
total = total + grade[i]
average = total / 10
```

If it makes sense to you that the two versions of the code above do the same thing, now we can make the big jump forwards: what we want to do are just the two lines

```
total = total + grade[i];
i = i + 1
```

over and over. And that's exactly what a for loop is designed to do! We'll make i our for loop counter variable, and update i as part of the for loop. So now we can rewrite the code as

```
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49}
double average = 0
double total = 0

// add up all the grades in the array
for (int i = 0; i < 10; i = i + 1) {
    total = total + grade[i]
}

average = total / 10
```

But what if we have more or fewer grades? Well the initialization of the array would have to change, but the rest of the code wouldn't get any longer. The only part that would have to change is the 10, which is the size of the array. But what if we are getting the grades from a method that doesn't tell us how many there are? We know how to ask the array itself for that value, so we could do

```
// empty square braces, size of array determined by method
int[] grade = getGrades()
double average = 0
double total = 0

for (int i = 0; i < grade.size; i = i +1) {
    total = total + grade[i]
}

average = total / grade.size
```

This will still work even if there are 100 or 1000 or more grades.

When we need to solve a problem involving an array, the answer is almost always to use a for loop, and that for loop almost always has the form

```
for (int i = 0; i < arrayVariable.size; i = i +1){
   do something using arrayVariable[i]
}
```

We would think of **arrayVariable[i]** as the **“current element of the array in this for loop”** if you can internalize that way of thinking about the loop, you'll always know what element you should be working on inside the loop: the current one, which means the one with i in the square braces!

So let’s print the elements of the grades array, but number them. If we are printing for a user to see, remember that we want to print using human numbers, not CS numbers, so we will need to increase the index number when we print

```
for (int i = 0; i < grade.size; i = i +1){
   print(“Grade #” + (i + 1) + “: “ + grade[i])
}
```

It’s the same for loop, but inside it, the thing we are doing with “the current element” grade[i] is printing it, with some nice formatting, including adding 1 to the index, so we get something like “Grades #1: 54” etc.

Random Numbers

Suppose I want to have a list of numbers in an array to test some code, but I don’t want to spend the time reading in values from a user, and I want to try lots of different values to make sure the code works for any situation. Many languages provide tools for generating random numbers.

Well… tools for generating *pseudo*-random numbers. Unless we are getting values from some truly random outside source[4], the “random” numbers in computing are always generated by some process that is complex enough to look random, but technically is not actually random, for instance it might do a calculation based on the number of milliseconds since the computer was turned on. This shouldn’t matter unless we are relying on the randomness for privacy or security reasons; in those cases we want to make sure that our results are random enough that some outside person couldn’t do the same calculation themselves to find out what our numbers were.

For our pseudocode, let's assume we have some methods for generating random values:

```
// generate a random double from 0.0 to 0.9999
double d = randomDouble()

// generate a random int - ANY int value is possible
int anyNum = randomInt()

// generate a random int from 0 up to but not including 10
int singleDigit = randomInt(10)

// generate a random true or false (50/50)
boolean coinFlipHeads = randomBoolean()
```

The randomDouble() method is probably the most common type of randomization tool provided by languages. It always generates a double from 0.0 to 0.9999, so you can think of it as 0 to *almost* 1. It turns out that this is enough to allow us to get to any range of doubles by doing some basic math[5]. If we want a larger range, we just multiply the result. If we want the range to start higher or lower than zero, we add to or subtract from the result.

```
// generate a random double from 0.0 to 99.999 (inclusive)
```

[4] For instance a company could generate random numbers by pointing a camera at a large number of lava lamps and using their constantly changing shapes to set the binary digits of a random number. Technically, with a complex enough fluid dynamics modeling program and the complete physical specifications of the lamps, the temperature in the room, etc, someone *could* generate these numbers too, but that would be a lot more random than the numbers we usually get.

[5] Doubles are used to represent real numbers in mathematics. There are as many real numbers between 0.0 and 0.999 as there are real numbers total. No, really. This is because the number of real numbers is what we call *uncountably infinite*; this particular class of infinity has this counterintuitive property, which I will happily show you the proof for if you come ask me about it in office hours. It’s a nice proof and doesn’t require any complex math.

To be fair, doubles only *represent* reals; the actual number of doubles is constrained by how many binary patterns we can fit in 64 bits, and so technically doubles work a little differently than reals, but the math still works out well enough for most purposes.

```
double d = randomDouble() * 100.0;
// generate a random double from -50.0 to 50.999 (inclusive)
double d = randomDouble() * 101.0 - 50.0;
```

Note that we could do the same kind of adjustment on the result of the version of nextInt that takes a parameter to move its range up or down.

Notice that there are two versions of randomInt(), the one with no parameters could give any int: large, small, positive, negative, zero. The one that takes an int parameter will always generate an int from zero up to but *not including* the value given, so if we give it 10, we get numbers from 0 to 9.

We can use these random generation methods to fill an array with random values.

Another common way we use random numbers with arrays is to choose a random value from a list. The idea is: first put the list of values you want to choose from into an array, then generate a random number that is a valid index in that array. You can then use the random index to extract a random value from the list:

```
// array of days of the week
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",
     "Friday", "Saturday", "Sunday"}

// generate a random index valid in the array
// a value from 0 up to but not including length of the array
int randIndex = randomInt(days.size);

// a randomly chosen day from the array
string randomDay = days[randIndex]
```

Sometimes when you are using random numbers to test your code and you discover that you have an error, it is useful to be able to test the code with the same sequence of "random" numbers over and over until you have fixed the problem.

Most random tools provide the ability to *seed* the random number generator, to provide a starting value for the "random" calculation used to generate the sequence of random numbers. If you give a seed, you will always get the same sequence, and then you can change the seed to test with a different sequence, knowing that if you find an error with those values, you'll be able to make changes and test again with the same values. We'll assume our pseudocode has a method randomSeed for doing this.

```
int seed = 989;
randomSeed(seed);
```

```
// this will print the same sequence of 100 numbers
// each time the program runs as long as it uses the
// same seed
for (int i = 0; i < 1000; i = i + 1) {
     print(randomInt())
}
```

Searching Arrays

Arrays give us a way to save large lists of data. Sometimes we want to just process every element, but sometimes we want to know if an array contains a specific value, and if so, where in the array (at what index) that value occurs. For simplicity we will assume that either the value only appears once or that we are always interested only in the first copy of a value.

Just Finding Whether a Value Is In The Array

Suppose that we want to know if there is a 0 in our array of grades. If we just need to report whether or not it is there at all, we could say

```
// mysterious method returns array with values
int[] grade = getGradeList()
// flag for whether we found a zero
boolean found = false

for (int i = 0; i < grade.size; i = i + 1) {
    // check whether the current value is 0
    // if so, update the flag
    if (grade[i] == 0) {
        found = true
    } // no else here!
}
// once we are done checking the whole array, report the result
if (found) {
    print("found a zero")
} else {
    print("no zeroes found")
}
```

We used a boolean called found to record whether or not we found a zero. Using a for loop as usual, we checked each element of the array. If the current element was a zero, we updated the found variable.

Notice that there is no else on the if inside the for loop. If the current element grade[i] *is* zero, then we want to update our boolean to true. But if it isn't, does that mean we should do something else? Update the boolean to false? NO! If the current element isn't zero, that just means we haven't found a zero *so far*. We still have the rest of the array to search.

In fact, adding an else that sets the boolean to false would be a disaster! If we haven't found our zero yet, then the boolean is still false, so setting it to false would be a waste of time. But what if we found the zero, set found to true, and then looked at the next number, which isn't zero… we'd be setting it back from true to false and losing our information!

The only time we can reasonably have an else, to report whether it was there or not, is after we have checked the entire array – after the end of the for loop.

Note that, if we really are only interested in the first occurrence, or know that there is only one, then we don't actually need to keep searching once it is found. So we could adjust the loop condition so that we stop once it is found.

```
// mysterious method returns array full of grades
int[] grade = getGradeList()
// flag for whether we found a zero
boolean found = false

// goes through whole array but stops if found
for (int i = 0; i < grade.size && !found; i = i +1) {
    // check whether the current value is 0
    // if so, update the flag
    if (grade[i] == 0) {
        found = true
    } // no else here!
}
// once we are done checking the whole array, report the result
if (found) {
    print("found a zero")
} else {
    print("no zeroes found")
}
```

Finding the Location of a Value

Suppose we want to know *where* in the array a value occurs. The location in the array is given by the index, so we will need to store the index where we found it.

```
int[] grade = getGradeList()
```

```
// let the user choose a value to search for
int seeking = prompt("What value to find?")

for (int i = 0; i < grade.size; i = i +1) {
    if (grade[i] == seeking) {
        // i is the index in the array where we
        // found the value, report it if found
        print("found a " + seeking + " at position " + i)
    }
}
```

In this case instead of just searching for zero, we made our code more general by getting a value, seeking, to search for from the user. Our loop looks much the same, but this time we said not just that we had found it, but the position in the array where we found it:, i. Remember that i is a computer science number; if we are really going to print to a user, we might want to print (i+1) instead. But actually, when searching for a location, we often want to store that in a variable to use later in the program:

```
int[] grade = getGradeList()

int seeking = prompt("What value to find?")

int foundAt = -1 // index position in the array

boolean found = false // whether we found it

for (int i = 0; i < grade.size && !found; i = i +1) {
    if (grade[i] == seeking) {
        foundAt = i // store the position
        found = true // store that we found it
    }
}

if (found) {
    print("found " + grade[foundAt] + " at " + foundAt)
} else {
    print(seeking + " not found")
}
```

In this version, we used the foundAt variable to store the index where we found the seeking value, so that it would be available after the for loop. We also used the found variable so that we would know whether we found it or not.

Note that we started foundAt as -1. We'll see why this is a good idea in a moment.

Finding Location and Recording Success in One Variable

It is much more common, instead of using both an int for position and a boolean, to just use the int. We can simply set the int to an impossible value for an index to start with, and if it still has that value after the loop ends, we know we must never have found the value. The standard value to indicate something was not found is -1, since that is not a valid index for any array:

```
int[] grade = getGradeList()

int seeking = prompt("What value to find?")

int foundAt = -1 // position starts with impossible value

// for loop goes through whole array
// but stops early if foundAt ever changes
for (int i = 0; i < grade.size && foundAt == -1; i = i +1) {
    if (grade[i] == seeking) {
        foundAt = i // changed to a real index
    }
}
// after loop, if foundAt is still -1 it was never changed
// so we never found it; if we did change it, it was found
if (foundAt != -1) {
    print("found " + grade[foundAt] + " at " + foundAt)
} else {
    print(seeking + " not found")
}
```

We started foundAt as -1. If we found the value we were seeking, we changed it to the current value of i. We can then use whether foundAt is still -1 both to help us end the for loop early and to check after the loop whether we found it.

Parallel Arrays

Each array can only store one type of data; we can have an array of ints and separately an array of strings, but one array can't store both ints and strings. If we have multiple lists of different types that are associated with each other, we can set them up as *parallel arrays*: multiple arrays of the same length where we use the same index across all arrays to group data.

Suppose that in addition to the array of grades, we have arrays of the first and last names of the students with those grades. Since there are the same number of first names and last names and grades, the arrays will be the same size, and we can use the same index to indicate the same person, grouping their first name, last name, and grade. So if we say:

```
int[10] grade // grades of students
String[10] fNames  // first names of students
String[10] lNames // last names of students

// all info about a single student across 3 arrays:
fNames[0] = "Jan" // student #0’s first name
lNames[0] = "Smith" // student #0’s last name
grade[0] = 85 // student #0’s grade

// all info about student #1, in three arrays
fNames[1] = "Gan"
lNames[1] = "Djones"
grade[1] = 90

// Stan Melchizadek got a 99
fNames[2] = "Stan"
lNames[2] = "Melchizadek"
grade[2] = 99
```

We are saying that there is a student whose name is Jan Smith whose grade is 85 – since all those values are at position 0 in their respective arrays, they go together. Similarly, Gan Djones got a 90 and Stan Melchizadek got a 99. You could think of the parallel arrays as each being a column in a table, and each index indicating a row in that table with all the information about a student.

Since parallel arrays must be the same size, we can use a single for loop to go through them all at the same time:

```
// use special initialization to create parallel arrays
int[] grade = {85, 90, 99, 80, 73}
String[] fNames = {"Jan", "Gan", "Stan", "Nan", "Ann"}
String[] lNames = {"Smith", "Djones", "Melchizadek", "Brown",
     "Whitley"}

// for loop can go through all three arrays at once
// since they are lined up and all the same size
for (int i = 0; i < grade.size; i = i + 1) {
    print(fNames[i] + " " + lNames[i] + " has " + grade[i])
}
```

If we search one array to find the position of a value, we can use the index we found to look up the related information in the parallel arrays. For instance, if we look up the last name of a student we can find the matching first name and grade.

```
// assuming we set up the three arrays already

prompt("Look up what surname?")
String lookFor = prompt("Look up what surname?")

int foundAt = -1
for (int i = 0; i < lNames.size; i = i +1) {
    // finding the location of the last name
    if (lNames[i] == lookFor) {
        foundAt = i
    }
}
// whichever position we found the last name,
// the matching first name and grade will be at
// the same position, in the other arrays
if (foundAt != -1) {
    print(fNames[foundAt] + " " + lNames[foundAt] + " has grade
     " + grade[foundAt])
} else {
    print("No such student")
}
```

Parallel arrays do rely on carefully maintaining the size and locations of data. If one of our arrays has data in the wrong order, everything breaks. When we learn about object orientation, we will learn how to group related data values into a single structure, and we can have single arrays of those programmer-created types instead of relying on parallel arrays.

Arrays with FirstEmpty

Arrays are a static sized data structure, which means that we have to choose the size of the array when we create it, and we cannot change the size later, we can only make a new array of a different size, and copy the data into the new one.

For this reason, we often create an array larger than the data we start with and then add values to it as the program runs; if we don't know how much data we will be dealing with, we will usually err on the side of making the array too large, because wasting some space is better than wasting a lot of time copying elements from one array to another.

Suppose we have an array that is already partially full, and a new value has just come in. We want to store the new value in the array, but not at a position where we have already stored a previous value.

One option would be to loop through the array until we find an empty spot… but there are several reasons this is a bad idea.

First: how do we know whether a spot is empty? If the array started with default values such as 0, false, or null, we could check for those, but false is certainly a valid possible value for a boolean array element, 0 is often valid for a number, and null could well be valid for any other type. We are back to the problem we had with sentinel values: this only works if there is some value that is not valid data.

But second: even if we do have some value we can use to indicate empty spots, looping through the array to find the first such spot every time we want to add is an *enormous waste of time*.

When we measure the time for a program to do a task, we measure not in terms of seconds elapsed, but based on how the number of steps is related to how much data there is: Suppose we are working with 10 data values in our array. Each time we search the array we have to look at on average (1/2)*10 items, and we have to do this 10 times, so (1/2)*10*10. If it were 1000 values, it would be (1/2)*1000*1000. So in general, if we are faced with n values, we have to do (1/2)*n^2.

We usually ignore the constant coefficient at the front: whether we are actually doing 18 lines of code to process each value, or just 3 lines, that number *(1/2) gets dwarfed by the number of values. So we ignore the (1/2) and just simplify this down to saying: doing it this way costs n^2 steps. That's not great if n is 10. That's catastrophic if n is 1000.

Especially when I tell you that we can get the same result in just n steps, if we use just one extra int-worth of space!!

We will keep an int variable as a sort of bookmark for the first spot in the array that is not currently occupied: the *first empty index*. We often call this variable something like firstEmpty or firstEmptyIndex or fei. Since array indices are always ints, it is always an int.

Suppose we are getting names from the user, and filling an array using a firstEmpty.

```
// the array so far, partially full
String[100] nameList
nameList[0] = "Abe"
nameList[1] = "Bea"
nameList[3] = "Cam"
```

```
nameList[4] = “Deb”
nameList[5] = “Eve”
// the firstEmpty so far
int firstEmpty = 6

final String QUIT = "QUIT"
String username = prompt("what name? " + QUIT + " to quit")

// while the user doesn’t quit
// and there is still space in the array
while (username != QUIT && firstEmpty < nameList.size) {
    // put the new name at the current first empty spot
    nameList[firstEmpty] = username
    // update firstEmpty so next name goes at next spot
    firstEmpty = firstEmpty + 1
    // get nextname from the user
    username = prompt("what name? " + QUIT + " to quit")
}

// if we ran out of space,
// let the user know that’s why we quit
if (firstEmpty >= nameList.size) {
    print("ran out of space");
}
```

The heart of this code is just putting the user name at the firstEmpty index in the array, and then incrementing the firstEmpty index, since that spot in the array isn’t empty anymore.

In this case, we wanted to read multiple names, so we did use a while loop, but notice that we didn't need another loop to actually add the value. Adding a new value to an array using a firstEmpty is one of those rare cases when solving a problem with an array does not need a for loop!

If we had been searching for the first empty spot, instead of using an int to keep track of it, doing that would have needed a for loop nested inside the while, which would have taken the very slow n^2 time.

Notice that we did check each time whether the firstEmpty had fallen off the end of the array by checking it against the array’s size. If we run out of space, then we either have to give up, or create a new, larger array, copy all the elements from the old array into the new, and continue from there (this is slow! So staring with an oversized array is smart.).

We put these values into the array because we have some use for them. If we need to process the values in our array we need to go through all of them, and that brings us back to using a for loop, but when we have an array with a firstEmpty, we don't usually want to process the whole array because there are probably a bunch of empty spots at the end of the array that we don't want to include in whatever process we are doing. So, the for loop for dealing with an array that has a firstEmpty has a condition that compares the counter to the firstEmpty, not to the size of the array:

```
// print the students in the array
// but not the empty spots at the end
for (int i = 0; i < firstEmpty; i = i +1) {
    print("Student #" + (i + 1) + ". " + nameList[i])
}
```

Copying and Resizing Arrays

In most languages, the array variable in fact stores the memory address of the first element of the array. So when we assign one array variable to another, what happens? In C++, this is an error. In Java, both variables can now be used to refer to the same array. In a couple languages, this does actually make a copy of the whole array.

But usually, to copy an array, we must make a new array of the same size, and copy each element over one by one.

```
// mystery method creates the original array
string[] original = getOriginalArray()

// the new array to copy into
// same size as original
string[original.size] destination

// these arrays function like parallel arrays
// so we can use one loop to go through them
for (int i = 0; i < original.size; i = i +1) {
    destination[i] = original[i]
}
```

If we planned poorly and ran out of room in our array, we can't make the array bigger so we need to make a new, larger array and copy all the elements over. One good guideline is to make the new array at least twice the size of the old.

```
// the original array
string[] original = getOriginalArray()
// the new array to copy into
// twice the size of original
```

```
string[2 * original.size] destination

// still just one loop, but we must use
// the smaller size, we are not filling up the
// larger array, just leaving the rest of it empty
for (int i = 0; i < original.size; i = i +1) {
    destination[i] = original[i]
}
```

Notice that if we were doing this for an array with a firstEmpty, the firstEmpty doesn't need to change. If it reached the size of the original array, that's okay, because that's now a valid index in the new bigger destination array and we can go on adding to it.

Multidimensional Arrays

In most languages, we can have an array where the elements are other arrays. In this case we just apply all the rules of arrays, twice – we have an array of arrays. We use two sets of square braces for two indices. In our pseudocode, we will assume the first set of square braces is the index in the array of arrays (which sub-array we want) and the second set of square braces is the index in that sub-array (which actual value we want).

We can picture a 2-dimensional array as a table with rows and columns (we can picture the sub-arrays as the rows or the columns, as long as within a given program we are consistent).

Let's picture a 2-dimensional array of ints

```
int[2][3] numtable
numtable[0][0] = 5
numtable[0][1] = 10
numtable[0][2] = 15
numtable[1][0] = 7
numtable[1][1] = 14
numtable[1][2] = 21
```

We could picture this as the table

	[0]	[1]	[2]
[0]	5	10	15
[1]	7	14	21

Storing a table of values is the most common use of a 2-dimensional array.

Anytime we need to process a 2-dimensional array, we need a for loop to go through each array (in this case, each array of ints) and a for loop to make sure we go through all of these arrays in our array of arrays. Therefore, we need nested for loops:

```
// go through the array of arrays of ints
for (int i = 0; i < numtable.size; i = i + 1) {
     // go through each int in the current array
     for (int j = 0; j < numtable[i].size; j = j + 1){
          print(numtable[i][j])
     }
}
```

Note that since numtable is an array of arrays, each element in numtable[i] is an array, so it also has a .size in our pseudocode. In less-friendly languages, we would have to keep track of that size in a separate variable. Because we set up this array to be 2 arrays of size 3 each, they are actually all the same.

In some languages, however, we can do this:

```
// some individual arrays
int firstRow[] = {1, 2, 3}
int secondRow[] = {1, 2, 3, 4, 5, 6, 7, 8, 9}
int thirdRow[] = {1, 2, 3, 4, 5}

int numtable[3][] // set up array of arrays with room for 3
     arrays (but no arrays in it yet)

// put the arrays in the array of arrays
numtable[0] = firstRow
numtable[1] = secondRow
numtable[1] = thirdRow
```

This array doesn't make such a nice table, the rows are all different lengths. But remember that arrays help us when our data is a list of the same type. If our data is a list of lists, an array of arrays may be useful, even if it doesn't look like a nice table, and we can still use nested loops to process it.

Arrays don't stop at 2 dimensions; we could have an array of arrays of arrays, and so on, just adding extra square braces. But unless we are dealing with data that is intrinsically multidimensional, such as a cube-shaped table of values, these are of limited use, particularly since everything in the table has to be the same type.