# Arrays

In the last chapter we learned about one type of container for storing data – the vector. In this chapter we learn about another, more basic, type of container – the array. Arrays are built into the language and are the most common data structure in computer science, as they are found in just about every programming language. Arrays are commonly used in situations where speed and efficiency are the most important factor in a program, though vectors are becoming more and more common for most programming scenarios due to their flexibility.

## Arrays Defined

An *array* is a set of contiguous memory locations that are reserved to store data of a single type. Arrays are fixed in size and cannot be resized (at least for now). We can view an array graphically as a two-column table where the first column stores the position (index) of an array element and the second column stores the data value of that element.

Here is how an array of 5 integers looks graphically:

|  |  |
| --- | --- |
| 0 | 8 |
| 1 | 12 |
| 2 | 34 |
| 3 | 89 |
| 4 | 0 |

We will mention this again but, like vectors, arrays are indexed starting with 0. Use this information proudly by annoying your friends by always ordering things starting with 0.

## Declaring Arrays

Arrays are declared similarly to vectors in that you must specify the data type for an array. You also must specify the size of the array because array are fixed size containers that can't grow or shrink after they have been declared.

The syntax template for declaring an array is:

*data-type array-name[number-of-elements];*

Here are some examples of declaring arrays:

int numbers[100];

string names[25];

double salaries[10];

The size of an array should be declared with an integer since the number of elements in an array has to be an integral value. Because the size of an array is an important piece of information you will need as you work with an array, it is conventional to declare an array with a constant that you can refer to later in your program.

Here are the examples above using constants in the declaration:

const int numElements = 100;

int numbers[numElements];

const int numNames = 25;

string names[numNames];

const int salariesSize = 10;

double salaries[salariesSize];

If you know what the elements of an array are at the time you declare the array, you can provide an initializer list with the elements. Here is the syntax template for this type of declaration:

*data-type array-name[number-of-elements] = {initializer list};*

Here are some examples of declaring arrays using initializer lists:

const int numGrades = 5;

int grades[numGrades] = {81, 77, 83, 92, 71};

const int numNames = 3;

string names[numNames] = {"Rebecca", "April", "Robert"};

const int numBools = 4;

bool flags[numBools] = {true, true, false, false};

## Accessing Array Elements

Array elements are accessed by specifying their index position within the [] operator. The following short program demonstrates how to add three integer array elements together:

#include <iostream>

using namespace std;

int main () {

const int numElements = 3;

int numbers[numElements] = {23, 1, 99};

int total = numbers[0] + numbers[1] + numbers[2];

return 0;

}

We can also access array elements to change the value of an element. Here is a code fragment that demonstrates how this works:

const int numElements = 3;

int numbers[numElements] = {23, 1, 99};

numbers[1] = 100;

## Accessing Array Elements with for Loops

When you want to access all the elements of an array, you should use a for loop. For a first example, we'll use a for loop to assign data to an array storing 5 course grades:

int main () {

const int numElements = 5;

int grades[numElements];

int grade;

for (int i = 0; i < numElements; i++) {

cout << "Enter a grade: ";

cin >> grade;

grades[i] = grade;

}

return 0;

}

Notice that the for loop starts at index position 0 and stops before it reaches the total number of elements in the array. This is because the numbering starts at 0 and in this case, the last element will be at index position 4.

This is a main reason why we always declare a constant with the array's size when using an array. It is a good software engineering practice to refer to a constant as the limit on a loop rather than a literal. This cuts down on mistakes and errors in your code.

You may be wondering what type of errors. Let's look at an example of accessing an array past the end of the array to see what happens.

Here is a program that creates an array of 5 integers and accesses the elements using an indexed for loop, but the loop goes past the last element of the array:

int main () {

const int numElements = 5;

int numbers[numElements] = {1,2,3,4,5};

for (int i = 0; i < 7; i++) {

cout << numbers[i] << " ";

}

return 0;

}

The output from this program should be:

1 2 3 4 5

But because the loop goes beyond the end of the array, we get this instead:

1 2 3 4 5 5 6

The program runs so there's not a syntax or even a run-time error, but the program produces wrong output, so this is a logic error.

When you declare a constant with the number of elements in an array and use it when accessing the array in a loop, you keep these types of errors from occurring.

If you are going to access all the elements of an array and want to ensure there will not be any errors in the access, use a range for loop.

The range for loop uses an *iterator* to keep track of the elements in an array, making sure that the loop starts at the first of the array and stops at the last element of the array. An iterator is an object that "points" to a container element. A special function sets the iterator to the first element of the array and a second function sets the iterator to a position just past the end of the array and the loop makes sure it doesn't get to that iterator.

Now having this high-level understanding of how the range for loop works, let's look first at the syntax template for this loop type before using it with an array. Here is the template:

*for (data-type variable-name : container-name) {*

*loop body;*

*}*

A variable is declared to hold each element of the array as it is accessed. The loop uses iterators behind the scenes to move from element to element, stopping with the last element.

Here is a short program that demonstrates how the range for loop works:

int main () {

const int numElements = 5;

int numbers[numElements] = {1,2,3,4,5};

for (int number : numbers){

cout << number << " ";

}

return 0;

}

The output from this program is:

1 2 3 4 5

For problems where you need to access every element of an array, you should use a range for loop rather than an indexed for loop. However, you can't use a range for loop when the array is used in a function definition for technical reasons that we aren't prepared to discuss at this point.

## Arrays as Function Parameters

An array can be used as a function parameter. When you do this, you must also include the size of the array as a second parameter so that the compiler will know where the array elements stop.

Let's start with a simple example. We want to compute the average of a set of grades that are stored in an array. We can write a function that has an array as a parameter, along with the size of the array. The function will total the elements of the array and then compute the average by dividing the total by the number of elements in the array.

Here is the function definition:

double gradeAvg(int arr[], int arrSize) {

int total = 0;

for (int i = 0; i < arrSize; i++) {

total += arr[i];

}

return static\_cast<double>(total) / arrSize;

}

The parameters are an array (note we must provide the data type) and a variable that represents the number of elements in the array. The array size must be left empty. The function computes the total by looping through the array and then returns the average (note also we convert the numerator to a double to make sure we get a floating-point result).

Now let's use the function in a program:

int main () {

const int numElements = 5;

int grades[numElements] = {81, 77, 72, 88, 91};

double average = gradeAvg(grades, numElements);

cout << "The average is: " << average << endl;

return 0;

}

Here is the output from this program:

The average is: 81.8

Remember that when you use an array as a function parameter you have to leave the size of the array empty and you have to pass the size of the array as an additional parameter.

## Creating an Array from a Function

Arrays cannot be return values from functions, though a pointer to an array can be (we cover pointers in a later chapter). We can, though, assign data to an array from inside a function because arrays are reference objects automatically, which means any changes to an array in a function are kept after the function call ends.

The following program demonstrates how to use a function to randomly generate data for an integer array:

#include <iostream>

#include <vector>

#include <random>

#include <ctime>

using namespace std;

void createArray(int arr[], int numElements) {

default\_random\_engine defEngine(time(0));

uniform\_int\_distribution<int> intDistro(0,100);

for (int i = 0; i < numElements; i++) {

arr[i] = intDistro(defEngine);

}

}

int main () {

const int numElements = 10;

int numbers[numElements];

createArray(numbers, numElements);

for (int n : numbers) {

cout << n << " ";

}

return 0;

}

The output from one run of this program is:

100 77 75 73 65 19 61 11 57 93

Again, this works because arrays are reference objects by default in C++. You don't have to pass an array by reference to a function to make permanent changes to the array from within the function.

## Two-dimensional Arrays

Some applications require that the data we store in our program be stored in the form of a table. An example is a list of students and their grades in a class. This data must be stored in a table where each student is represented by a row in the table and the grades make up the columns of the table.

We can use an array to store tables of data in C++. A table that has a set of row and columns is called a *two-dimensional array*. A two-dimensional array is declared with two sets of brackets for the rows and columns. Here is the syntax template for a two-dimensional array declaration:

*data-type array-name[number-of-rows][number-of-columns];*

Here are some examples of declaring two-dimensional arrays:

const int numRows = 3;

const int numCols = 3;

int grades[numRows][numCols];

const int nRows = 2;

cont int nCols = 4;

string records[nRows][nCols];

Data are assigned to two-dimensional arrays by referencing a row and a column, like this:

grades[0][0] = 88;

grades[0][1] = 83;

grades[2][2] = 99;

grades[3][1] = 100;

If you know in advance what data are going to be stored in the array, you can use an initializer list, as in this example:

const int numRows = 2;

const int numCols = 2;

int grades[numRows][numCols] = {{81, 77}, {91, 98}};

Accessing all the elements of a two-dimensional array usually requires the use of a nested for loop. A nested for loop is when the body of an outer for loop consists of another for loop. The outer loop controls access to the rows of the array and the inner loop controls access to the columns of the array.

The following example creates a new two-dimensional array and displays the data row-by-row:

int main () {

const int numRows = 3;

const int numCols = 3;

int grades[numRows][numCols] = {{71, 82, 93},

{88, 85, 91},

{74, 66, 78}};

for (int row = 0; row < numRows; row++) {

for (int col = 0; col < numCols; col++) {

cout << grades[row][col] << " ";

}

cout << endl;

}

return 0;

}

Here is the output from this program:

71 82 93

88 85 91

74 66 78

Notice that between the inner loop and the outer loop there is a newline statement that separates the two rows of data.

Now that we've seen this pattern for processing a two-dimensional array, let's use it to do some calculations. We can calculate the test average for each student by working across the columns, adding each column to a total variable and then dividing by the total number of columns to get the average. The inner loop controls adding up the columns and the outer loop moves from row to row, just as we did when we displayed the data in the array.

Here's the program:

int main () {

const int numRows = 3;

const int numCols = 3;

int grades[numRows][numCols] = {{71, 82, 93},

{88, 85, 91},

{74, 66, 78}};

int total = 0;

double average = 0.0;

for (int row = 0; row < numRows; row++) {

for (int col = 0; col < numCols; col++) {

total += grades[row][col];

}

average = static\_cast<double>(total) / numCols;

cout << "Student " << row+1 << " average: " << average;

total = 0, average = 0.0;

cout << endl;

}

return 0;

}

Here's the output from the program:

Student 1 average: 82

Student 2 average: 88

Student 3 average: 72.6667

Notice that between the inner loop and the outer loop the variables total and average are reset. This must happen for the calculations to be correct. If we didn't set these variables back to 0, the totals would continue from student to student, leading to numbers that are too high.

## Two-dimensional Arrays as Function Parameters

Two-dimensional arrays can be used as function parameters just as one-dimensional arrays can, but there is one special notation you must make in the function definition. You must pass the number of columns of the array to the function so the function will know how large the array is. You will also want to pass the number of rows, but the number of columns must be passed in a specific way, as shown in this syntax template:

*return-type function-name(data-type array-name[][number-of-columns], …) {*

*function body;*

*}*

Let's write a function that we can use to display the contents of a two-dimensional array. Here is the function definition:

void show2DArray(int arr[][3], int numRows, int numCols){

for (int row = 0; row < numRows; row++) {

for (int col = 0; col < numCols; col++) {

cout << arr[row][col] << " ";

}

cout << endl;

}

}

Here is a program that uses the function:

int main() {

const int numRows = 3;

const int numCols = 3;

int grades[numRows][numCols] = {{71, 82, 93},

{88, 85, 91},

{74, 66, 78}};

show2DArray(grades, numRows, numCols);

return 0;

}

Here is the output from this program:

71 82 93

88 85 91

74 66 78

## Glossary

**array**: a contiguous set of memory locations dedicated to storing data of the same data type

**two-dimensional array**: a contiguous set of memory locations that store data of the same type in a table of rows and columns

## Exercises

1. Declare and initialize an integer array to store 20 randomly generated integers. Use a range for loop to display the numbers stored in the array.
2. Design and implement a function that displays an array of twenty or more elements so that the display is made up of rows of 10 elements per row. Use an indexed for loop in your function definition. Use the function in a program.
3. Declare and initialize an array of twenty randomly generated strings. Use the internet to figure out how to create a random string. Display the strings from the array in order using a range for loop and then display the strings in the array backwards using an indexed for loop.
4. Design and implement a function that finds the median value from an array of at least 100 elements, generated randomly. Use the function in a program to report the median value from that array.
5. Declare and initialize a two-dimensional array that contains a 5-by-5 table of double values that represent grade averages. Design and implement a function that finds the lowest value in the array and a second function that finds the highest value in the array. Use the functions in a program.