

CS 161: Arrays – Chapter 8

This week, you will learn about arrays. An array is a data structure that contains a sequence of data items of the same type. You will learn how to declare arrays, populate them with data, access the data items and pass them to functions.

Much of the data you will work with as a software engineer are array-like such as lists and tables. Because of this, arrays are very important in almost all software and are part of every language.

Purpose of these notes

These notes provide an overview of the chapter 8 material. Please read the book to see complete code examples and more in depth discussions of the topics.

Notes about these notes

To keep things brief, some lines are left out of the code samples in these notes. The goal is to show what was necessary to illustrate the point. You will sometimes see “...” in the examples which means “some necessary code for a complete program are not shown here”.

Additionally, the order of the topics in these notes follow the presentation order in the book for the most part but some items such as strings and arrays are placed in a different place in these notes as compared to the book.

Contents

| | |
|--|----|
| Purpose of these notes | 1 |
| Notes about these notes..... | 1 |
| Arrays | 4 |
| What is an array? | 4 |
| How do you declare an array? | 4 |
| Method 1: declare with size and no initializer..... | 4 |
| Method 2: declare with size and initializer..... | 4 |
| Method 3: skip size and just use initializer | 4 |
| How do you access the elements of an array? | 5 |
| Arrays start at index 0 and go to length -1 | 5 |
| Decisions that you as the programmer can make | 5 |
| Is this a good thing to do | 5 |
| Understand how “[” and “]” are used..... | 5 |
| During array declaration | 5 |
| To define the subscript value..... | 6 |
| How do you put data into an array? | 6 |
| How big of an array should you create? | 6 |
| Arrays allow random access | 7 |
| Default values for arrays..... | 7 |
| Passing arrays to functions | 8 |
| Array are automatically passed by reference | 8 |
| Arrays of class objects..... | 9 |
| How are class objects initialized when declaring an array? | 10 |
| Calling a member function of class object stored in an array | 11 |
| Arrays of structures..... | 11 |
| Two dimensional arrays | 12 |
| Declaring a 2D (2 dimensional) array..... | 12 |
| Accessing data values | 12 |
| What should be the row element and the column element? | 13 |
| Passing 2D arrays to functions..... | 13 |
| Arrays with three or more dimensions..... | 13 |

| | |
|--|----|
| Indexing of higher dimensional arrays..... | 14 |
| Passing higher dimensional arrays to functions | 14 |
| Vectors | 14 |
| Why use a vector over a 1D array? | 15 |
| Declaring a vector | 15 |
| How to store and retrieve data in a vector..... | 16 |
| Checking vector size..... | 16 |
| Using a vector's ability to grow in size..... | 16 |
| Other vector functions..... | 17 |
| Parallel Arrays | 17 |
| Strings and arrays..... | 18 |

Arrays

What is an array?

An array is a data structure or data container that allows you to access and store a linear sequence of data items of the same type. Unlike when you declare an int or a float, which allows you store a single item, you can create arrays to hold many items that you can access from a single array variable.

My shopping list → a list of strings → `string myShoppingList[6];` // array declaration

| |
|----------------------|
| Bread |
| Milk |
| Inversion IPA |
| Mirror Pond Pale Ale |
| Eggs |
| Turkey |

How do you declare an array?

You can create array variables through three ways.

Method 1: declare with size and no initializer

```
int myExampleArray[6]; // create an array that will hold 6 int values
```

This type of declaration creates an empty array. The compiler saves spaces for your data but it is up to you to fill in the data later.

Method 2: declare with size and initializer

When you declare an array, you can specify initial values by providing an initializer. An initializer is a list of values of the proper type between a set of curly brackets.

```
int myExampleArray2[6] = {12, 5, 67, 11, 88, 91};
```

Note: the initializer can contain a number of items up to the size of the array. If there are less items in the initializer than the size of the array, the remaining items are set to null.

```
int myExampleArray2[6] = {12, 5, 67, 11}; // array contains 12, 5, 67, 11, 0, 0
```

Method 3: skip size and just use initializer

```
int myExampleArray2[] = {12, 5, 67, 11, 88, 91, 2, 3};
```

In this case, the compiler will create an array of the proper size to hold all of the values in the initializer. Since there are 8 items in {12, 5, 67, 11, 88, 91, 2, 3}, the array will be of size 8.

How do you access the elements of an array?

Arrays are a linear sequence of data items. You use what are called subscripts to access individual elements of the array.

```
cout << myExampleArray[5] << endl; // [5] is the subscript
```


subscript

Arrays start at index 0 and go to length -1

It may seem foreign to you at first but you use the subscript value of 0 to access the first value of an array

```
int testArray[4] = {4, 64, 5, 1001};
```

```
int testValue = testArray[3];
```

What will be the value of testValue?

Since the value of the subscript is 3, you are accessing the 4th element of the array which is 1001;

Decisions that you as the programmer can make

If the item you are putting in an array has a really strong and natural 1 to n type of property, you can declare your array to be one item longer than needed and then decide to not use the first subscript in the array.

```
int monthlyData[13];
```

Here the monthlyData array has space for 13 values and you as the programmer can decide to only use subscripts 1 to 12.

Is this a good thing to do

There is no right or wrong answer here. Over time, you will get used to subscript 0 denoting the first value of the array and thinking that if I want the nth value in the array, I need to use a subscript value of n-1.

Probably the biggest drawback of designing your program to use subscript 1 to hold the first value of the array is that other programmers might work on your code in the future and will probably automatically assume that subscript 0 holds the first value.

Understand how “[” and “]” are used

Square brackets are used two different ways when working with arrays.

During array declaration

When you write

```
float testScores[5];,
```

the [] are used to declare the size of the array. In this case, the array will hold 5 float values.

To define the subscript value

```
float aValue = testScores[2];
```

```
testScores[1] = 100.0;
```

How do you put data into an array?

As you saw above, you can add data into an array by using an initializer. Initializers are nice because you can add all of the data in one statement but you need to know what the data will be when you write the program. Note: you can only use the initializer method at the time you declare the array.

If you don't put in data using an initializer, you will have to set the data by accessing each individual element of the array. Often, programmers create loops to write data into an array (see example in the following section).

How big of an array should you create?

The array declaration methods above require a fixed array size. This is not a problem if you know exactly how many elements you need to store. For example, if you wanted an array to store the number of days in each month, you could create an array like `int daysInMonth[12]`.

However, you will find that in many applications where you want to use an array you will not know the exact numbers of items that you will want to store in your array. For example, you decide to use an array to store test scores for students in cs161 so you can compute the test average. Since you don't know how many students will be in the class in any given quarter, you can define the size of the array for some worst case or upper bound.

```

#define MAXSTUDENTNUMBER 300 // assume that there will never be more than 300 students

int main()
{
    int testScores[MAXSTUDENTNUMBER];
    ...
    int currentIndex = 0;
    string userInput;
    int testScore;
    while(currentIndex < MAXSTUDENTNUMBER) // make sure you don't exceed max array size
    {
        cout << "enter test score: ";
        cin >> userInput;
        // check to user is done; this is denoted when they enter 'q'
        if(userInput != "q")
        {
            testScore = atoi(userInput.c_str());
            testScores[currentIndex++] = testScore;
        }
        else // user entered "q" so quit input process
            break;
    }
    if(currentIndex >= MAXSTUDENTNUMBER) // user reached maximum number
        cout << "you have reached the max number of test scores that you can enter" << endl;
    ...
}

```

Word of caution: setting an upper bound/max value allows you to handle the case where you don't know exactly how big to make your array. However, choose an upper bound wisely. Even if you don't store data in part of the array, the program will declare or hold on to that memory until the program is completed so if you declare some super huge number so you will never ever run out of array space, your program will always be a huge memory hog.

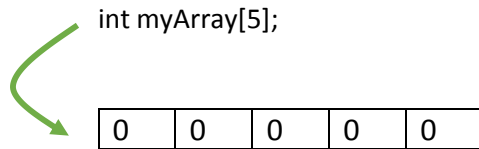
Arrays allow random access

You enter or get data from an array in any order. You don't have to start with myArray[0].

It can be helpful to have a variable to store the last used index of an array

Default values for arrays

When you declare an array without an initializer, the array elements are set to default values which is 0 for fundamental types such as int, float, and bool.



Passing arrays to functions

To declare an array as a parameter in a function, you list the type followed by “[]”. For example, in the following array, the first parameter is an array of int data.

```
void functionThatHasArrayParameter(int [] myArrayData) {  
    ... // body of function  
}
```

While the above function declaration is perfectly correct, since you can’t determine the size of the array from the array itself in the function, you often need to pass the array length/size as another parameter.

```
void functionThatHasArrayParameter(int [] myArrayData, int arrayLength) {  
    ... // body of function  
}
```

Array are automatically passed by reference

With the other data types you have learned about, you pass a variable by reference by using the “&” symbol such as

```
void aFunction(int &referenceInput);
```

However, arrays are automatically passed by reference. In fact, there is no way to pass by value. Since arrays can be quite large, it would be very inefficient to create copies of arrays to pass them by value to functions.

What is the significance of all of this?

Since arrays are passed by reference, if you modify the array inside the function, you are modifying the actual data and any use of the array after the function returns will see the modifications made in the array.


```
void aFunction(int [] data, int length)
{
    data[2] = 0;
}

int main()
{
    int myData = {5, 5, 5, 5};

    cout << myData[2] << endl; // outputs value of 5

    aFunction(myData);

    cout << myData[2] << endl; // outputs value of 0
}
```

Use the const keyword to protect arrays from changes in functions

If you don't need to modify the array in a function, use the const keyword in the function declaration. If you do this and write code that modifies the array data within the function, you will get an error at compile time. This will prevent you from accidentally modifying the array values.

```
void functionWithReadOnlyAccess(const int[] myData, int length);
```

Arrays of class objects

All of the above examples have used the C++ simple data types such as float and int. However, you can also create arrays to hold complex data types such as class objects.

To create an array for a class data type, you simply use the class name in the array declaration statement.

```
class Circle
{
    private:
        double radius;
        string label;
    public:
        Circle() // default constructor
        {
            radius = 1.0;
            label = "";
        }
        Circle(double r)
        {
            radius = r;
            label="";
        }
        Circle(double r, string l)
        {
            radius = r;
            label = l;
        }
        ...

        double findArea() { // member function
            return 3.14 * pow(radius, 2);
        }
        ..
    }
};

int main() {
    Circle circleData[100]; // create an array to store 100 Circle objects
    ...
}
```

How are class objects initialized when declaring an array?

Declaring an array with no initialization list

When you declare an array of class objects, each element in the array is filled by the results of the classes' default constructor.

```
Circle circleData[100]; // each Circle object set to results of default constructor
```

Declaring an array with initialization list of single values

You can provide an initialization list to provide a single value to the object's constructor.

```
Circle circleData[3] = {2.0, 3.5, 5.0};  
// each Circle object set to results of constructor that takes single argument
```

Declaring an array with initialization list of function calls

If you want to fill an array with results of constructors that take more than one argument, you must use constructor function calls.

```
Circle circleData[2] = { Circle(2.0, "first circle), Circle(5.0, "second circle") };  
// calling 2 parameter constructors
```

Calling a member function of class object stored in an array

Continuing the above example

```
circleData[3].findArea();  
// call the Circle findArea member function for 4th Circle object  
// in the circleData array
```

Arrays of structures

Arrays of structures are handled the same way as arrays of objects.

```
struct BookInfo  
{  
    string title;  
    string author;  
};
```

```
BookInfo bookData[100]; // create an array to store 100 BookInfo structures
```

If you want to set initial values of the structures to be different than the variable defaults, you will need to add a constructor to your structure and declare the structure array the same as done with class objects and initializers.

After the bookData array is loaded with data, you can access individual structures and associated member variables.

```
cout << bookData[2].title << endl; // output title of 3rd book in bookData array
```

Two dimensional arrays

The above discussion of arrays has focused on what are also known as one-dimensional arrays. The data can be thought of as a linear list of data objects or a single row; however, you can also create arrays in two or more dimensions.

Two dimensional arrays allow you to store data in what is best conceptualized as a row and column format.

| | 1 | 2 | 3 | 4 |
|---|-----|------|------|------|
| 1 | 123 | 11 | 3592 | 1956 |
| 2 | 40 | 351 | 3 | 26 |
| 3 | 290 | 7000 | 41 | 979 |

Declaring a 2D (2 dimensional) array

```
long myExample2DArray[3][4];
// declare a 3 x 4 array to store the data in the table above
```

Using an initializer list

Both of the following are equivalent. Data are listed in row 0, row 1, ... to row n-1 order.

```
long myExample2DArray[2][3] = { 1, 2, 3, 4, 5, 6 }; // row breaks calculated by number of
// columns
long myExample2DArray[2][3] = { {1, 2, 3}, {4, 5, 6} }; // explicit definition of row breaks
```

These result in the following

| | Col 0 | Col 1 | Col 2 |
|-------|-------|-------|-------|
| Row 0 | 1 | 2 | 3 |
| Row 1 | 4 | 5 | 6 |

Accessing data values

You access an element in a 2D array by providing a subscript for both the row and column part of the array.

```
long aValue = myExample2DArray[3][1];

myExample2DArray[0][29] = 1000000;
```

What should be the row element and the column element?

If you needed to store 10 objects that each have 10000 elements, should you create a short and wide array or a tall and narrow array? The answer is that it does not matter; both are equal in the eyes of the computer/compiler.

```
long tallNarrowArray[10000][10];
```

```
long shortWideArray[10][10000];
```

We suggest to choose subscript order by what seems most natural. Choose the property for the first subscript that most naturally comes first. For example, I want to store data by month and day. To me, I think of month first such as April 6, so I create my array as follows:

```
int monthByDayData[12][31];
```

I chose 31 for the size of the second dimension as months can have up to 31 days

```
monthByDayData[3][5] = 100; // April 6
```

Passing 2D arrays to functions


In review, when declaring a function with a 1D array parameter, we did it as follows:

```
void functionWith1DArray(int[] arrayData, int arrayLength);
```

To declare a function with a 2D array parameter, you must specify the size of the second dimension:

```
void functionWith2DArray(int[][31] monthDayData);

int main() {
    int monthDayData[12][31];
    ... // code to fill array not shown
    // call function
    functionWith2DArray(monthDayData);
}
```



Note: you can use the const keyword with any array so you can use them on your 2D arrays.

Arrays with three or more dimensions

Arrays are not limited to 1 or 2 dimensions. You can declare as many dimensions as you need for your data design.

1D array: think of as list
 2D array: think of as table
 3D array: think as cube-like but does not need to be square
 4D array: maybe harder to visualize but can be created
 ... and so on

We used month and days as an example of a 2D array data set. If we now want to track that data over years, let's assume we will reference the data as year, month, and day of month.

```
const int NUMYEARS 100;
int dataByYearMonthDay[NUMYEARS][12][31];
```

If you need to create higher dimensional arrays, you just need to specify additional subscripts in the array declaration.

```
int sixDimensionArray[10][10][10][10][10][10];
```

Indexing of higher dimensional arrays

You must specify a subscript value for each dimension of the array to access a single element of the array:

```
int dataByYearMonthDay[NUMYEARS][12][31];

dataByYearMonthDay[10][3][5] = 100;
```

Passing higher dimensional arrays to functions

When specifying higher dimensional arrays as parameters for functions, you must define the size of every dimension except the first dimension.

```
void processYearMonthDayData(int[][12][31] myYearMonthDayData);

int main() {
    int dataByYearMonthDay[NUMYEARS][12][31];

    processYearMonthDataData(dataByYearMonthDay);
    // pass in 3D data as argument
}
```

Vectors

A vector is data type that allows you to store and manipulate 1D data. It is similar to a 1D array but includes many built in functions that you would have to write yourself if using a 1D array.

Vectors are part of the Standard Template Library (STL). The STL provides many useful data types and algorithms to use in your C++ programs. Use of the STL allows you to focus your programming on the unique parts of your program.

Why use a vector over a 1D array?

As we will show below, the primary advantage of a vector is that vectors are not fixed in size or length. You can add items to the end of a vector and the vector will automatically allocate more memory if needed to store the new item.

Declaring a vector

First, you must include the vector header file when using a vector in your program.

```
#include <vector>
```

Second, there are four common ways to declare a vector. The code for all of these examples starts with

```
vector<typename> nameOfVariable ...
```

The <typename> part of the definition is where you specify what type of data you will store in the vector. For example

```
vector<int> myVectorOfInts ....    // vector will store int values
vector<double> myVector ....      // vector will store double values
```

You can also create vectors to store class objects and structures but we will not use that capability this week.

Method 1: declare with no initial size.

```
vector<float> exampleVector;
```

Method 2: declare with initial size

```
vector<int> exampleVector(10);    // note the use of ( and ) instead of
                                   // square brackets
```

Method 3: declare with initial size and default value

```
vector<int> exampleVector(10, 2); // size is 10 and every value is
                                   // initialized to the value of 2
```

Method 4: declare and initialize with values from another vector

```
vector<int> vectorOne(10, 2);      // size of 10 initialized to 2
vector<int> vectorTwo(vectorOne);
```

vectorTwo copies values from vectorOne so it is also of size 10 with values = 2

How to store and retrieve data in a vector

If your vector has space allocated for data which is the case for all of the initialization methods except where you don't specify an initial size, you can access vector elements just like a 1D array by using subscripts.

```
vector<float> myFloatVector(10); // initial size is 10
float f = myFloatVector[5];      // get the 6th value in the vector
myFloatVector[2] = 3.14;         // set the 3rd value to 3.14
```

You can also use the `vector::at` function to retrieve values

```
float f = myFloatVector.at(5);
```

Just like arrays, bad things happen when you try to set or retrieve values outside of the current vector bounds.

Checking vector size

To avoid these bad things, you can get the size (the number of elements in the vector) of vector by calling the `vector::size` function.

```
vector<double> myVector(55);
int lengthOfVector = myVector.size(); // lengthOf Vector should = 55
```

Using a vector's ability to grow in size

You can increase the size of a vector in two ways after it is initially declared.

Adding values to the end by `vector::push_back` function

You can add values to the end of a vector by calling the `push_back` function.

```
vector<double> myVector(10); // size of vector is 10

myVector.push_back(1.23);    // size of vector is now 11

cout << myVector[10] << endl; // should output 1.23
```

Increase size through the `vector::resize` function

If you need additional elements in your vector, you can increase the size by calling the `resize` function.

```
vector<string> myVector(10); // size is 10

vector.resize(20);           // size is now 20
```


You can also call `resize` and provide a 2nd argument to set the initial value of the added elements.

```
vector.resize(25,100);  
// now, the length is 25 and the elements from 19 to 24 have the value  
// of 100.
```

If you call `resize` with a value less than the current size, the vector is resized to the new value and the data beyond the new size are lost.

Other vector functions

We have discussed the functions `at`, `size`, `push_back`, and `resize`. There are other functions for vectors that are discussed in the book on page 567. The function `pop_back` is useful for removing the last element of the vector and decreasing the size by one.

Parallel Arrays

The book discusses parallel arrays. Parallel arrays are the use of two or more arrays to store multiple data values for individual objects/things/entities.

What is an object/thing/entity? Basically something that can have features and that we can identify by some numerical index.

In a parallel array, we use the subscript to identify the object such as employee id, product id, day of week, or month of year.

Then, each array stores some data about the object. Let's consider the case where we want to track sales, returns, and net profits by month.

To do this, we create three arrays:

```
float monthlySales[12];  
float monthlyReturns[12];  
float monthlyProfit[12];
```

To store data for the month of February, we could write:

```
monthlySales[1] = 50000;  
monthlyReturns[1] = 1200;  
monthlyProfit[1] = 22000;
```

Note: it is easy to get confused by the array indexing. February is the second month but the proper index is 1. If I were writing a full program for this example, I would probably declare month constants such as

```
const int FEBRUARY 1; // allow me to do things like monthlyProfit[FEBRUARY] = 22000;
```

While parallel arrays do allow you to keep track of multiple types of data for a given object, in practice you will probably find it easier to create arrays of structures or classes if you need to access multiple features for an object.

Strings and arrays

Variables of the C++ string type are stored internally as arrays of characters.

```
string greeting = "hello";  
  
char letter;  
  
letter = greeting[2]; // letter = 'l'
```

| | | | | |
|-------------|-------------|-------------|-------------|-------------|
| 'h' | 'e' | 'l' | 'l' | 'o' |
| greeting[0] | greeting[1] | greeting[2] | greeting[3] | greeting[4] |

Unlike like the other arrays covered in chapter 8, you can easily get the length of the string array by using the string length function

```
int length = greeting.length(); // length = 5 bytes
```

Special Note: the C++ string.length function returns the number of bytes used to store the string. Some character encodings use more than one byte per character so don't always assume that the value of the length function always equals the number of actual characters.