C/C++ PROGRAMMING

Introduction to Arrays

```
int score[10] = {9, 8, 10, 9, 7};
```

Contents	9	8	10	9	7	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

Dan McElroy

Updated March 2020

Hello programmers. This discussion introduces arrays as used by C and C++.

Definition

An **array** can hold multiple pieces of data of the same type. The position in an array is called an **index**. The first position in an array is at index 0;

Declare an array of 10 integers:

```
int score[10];  // score can hold 10 integers
```

The index values for the array go from 0 to 9.

An **array** can hold multiple pieces of data of the same type. The position in an array is called an **index**. The first position in an array is at index 0;

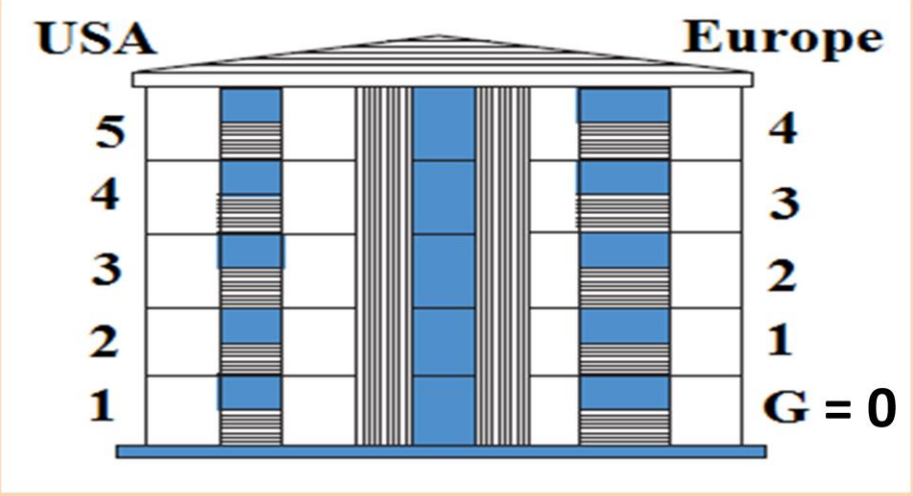
Here is an example: Declare an array of 10 integers:

```
int score[10];  // score can hold 10 integers
```

The index values for the array go from 0 to 9.

Since the index starts at 0, index values for an array of 10 elements are 0, 1, 2, 3, 4,5 ,6 7, 8, and 9. Trying to use a negative index value or a value of 10 or higher would be an illegal attempt to access locations that are not in the array.

Array Index Starts from Zero

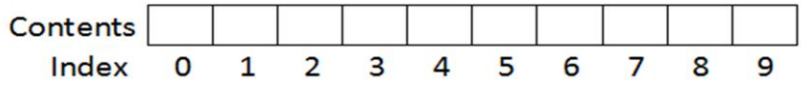


Array Index Starts from Zero. It is most common to start counting from zero because the first index (position) in an array starts with zero.

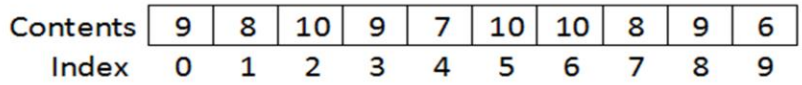
This may be confusing for most Americans who count the ground floor as the first floor, but the rest of the world counts the first floor as the floor above ground. Index values start at zero because it is easier to internally to compute the address of individual elements by adding the index number times the element size to the starting address of the array.

Declaring an Array

int score[10];



int score[10] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};



The first example shows only that an array of 10 integers has been declared. The square brackets [ ] are used to identify an array. Enough room in memory is reserved to store 10 integers.

If the array is declared as an automatic variable inside a block of curly-braces { } then the contents of the array are not initialized. This means that they are full of garbage data that happened to be in those memory cells before the array was declared. It is up to the programmer to place some known value in the array elements before trying to use them, increment them, etc. If the array is declared with the static keyword, or if the array is declared as global data outside of any block of code, then the contents of the array are initialized to zeros.

The second example shows an array being declared with 10 integers, and the contents of the array are being initialized are being initialized at the same time. The curly-braces are placed around the initializer data, and each element is separated by a comma.

Default Array Size

If an array is initialized when it is declared, it is not necessary to give the size of the array.

```
int score[ ] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};
  // is the same as
int score[10] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};
```

Contents	9	8	10	9	7	10	10	8	9	6
Index	0	1	2	3	4	5	6	7	8	9

If an array is initialized when it is declared, it is not necessary to give the size of the array. The size of the array is determined by the number of elements listed inside the curly-braces.

Array Size

```
int score[10];
int s1 = sizeof(int);       //  s1 = 4 for most compilers
int s2 = sizeof(score);   //  s2 = 40

int count = sizeof(score) / sizeof(int);  // count =4
```

The C and C++ compilers need to know the size of an array when it is declared. Here is an array of 10 integers. For most compilers, the size of an integer is 4 bytes, so the size of the array would be 40 bytes. The sizeof operator can be used to determine the size in bytes for data types, arrays, and other things.

The sizeof operator can also be used to determine the number of elements in an array by taking the size of the array and dividing it by the size of each element in the array.

Partial Initialized Array

```
int score[10] = {9, 8, 10, 9, 7};
```

Contents	9	8	10	9	7	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

If the size of an array is declared and the array is initialized with fewer values than the size of the array, the remaining elements are set to zeros.

Access an Element in an Array

```
int score[10] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};
```

Contents	9	8	10	9	7	10	10	8	9	6
Index	0	1	2	3	4	5	6	7	8	9

```
int x = score[4];  // x gets a 7 from index position 4
int y = score[5];  // y gets a 10 from index position 5
int z = score[1];  // z gets an 8 from index position 1
```

The index MUST be an integer.

An individual element in an array can be accessed by giving the name of the array and placing the index number of the element inside square brackets. In these examples,
x gets a 7 from index position 4
y gets a 10 from index position 5
z gets an 8 from index position 1
The index into an array must be an integer data type, which in C and C++ includes the char data type.
An error will happen if trying to access elements before or after the end of the array.

The Index Can Be an Expression

```
int score[10] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};
```

Contents	9	8	10	9	7	10	10	8	9	6
Index	0	1	2	3	4	5	6	7	8	9

```
int n = 3;
int x = score[n+1];  // x gets a 7 from location 4
```

The index can be an expression. In this example, n is a 3, so n+1 is a 4. Therefore, x gets a 7 from location 4.

An Index Can Be Incremented

```
int score[10] = {9, 8, 10, 9, 7, 10, 10, 8, 9, 6};
int total = 0;
int i;
for (i=0; i<10; i++)  // for index positions 0 through 9, not 10
   total += score[i];
int avgScore = total / 10;
cout << "Average = " << avgScore;
```

This is a very common way the **for** statement is written when working with arrays.

Notice that the index i starts at zero and goes up to but not including 10. The test statement in the for loop is i < 10; This for loop is being used to compute the total of all the values that are in the array, and then compute and display the average score.

Now that we are working with arrays, it might make more sense why it is most common to count for loops starting at 0 and going up to but not including the highest value.

Use a Constant for the Array Size

```
const int CLASS_SIZE = 10;
int score[CLASS_SIZE];
int i;
for (i=0; i<CLASS_SIZE; i++)
{
   cout << "Enter a score: ";
   cin  >> score[i];
}
```

Here is a fantastic reason to use constants. The same constant is used when declaring the size of the array, and also used inside the for loop to increment through the array.

If CLASS_SIZE ever changes, it will be changed in every location that it is used.

What would happen if you just used the number 10 and forgot to change one of them. Or even worse, if you wanted to change the size of the array from 10 to 12 and then used find and replace in the editor to change all 10's to 12's, a value of 100 would also change from 100 to 120, and a value of 7108 would be changed to 7128.

This is probably not what would be desired, but unintended find and replaces have happened to me.

Out of Range Condition

```
int score[10];
int i;
for (i=0; i<=10; i++)
{
   cout << "Enter a score: ";
   cin  >> score[i];
}
```

In this case, the loop moves the index from 0 to 10 because the test in the loop is i<=10 which includes the index value of 10.

The last position in the array is 9. The program will either report an error, or even worse, store something in memory past the end of the array, probably on top of some other data without any warning. It may be hard to find out why data in a different part of the program got changed when that part of the program looked OK.

Multidimensional Arrays

Arrays can have more than one dimension.

```
// one student, five scores
int studentScores[5];

// nine students, five scores each
int classScores[9][5];

// three classes of nine students, five scores each
int schoolScores[3][9][5];
```

Arrays can have more than one dimension.

Here is an example of a single-dimensional array. It is created for one student with five scores.

```
int studentScores[5];
```

And now, lets create a double-dimensional array that can hold nine students with five scores for each student.

```
int classScores[9][5];
```

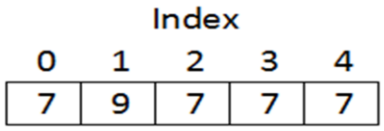
Suppose I was teaching three different classes with nine students in each class, and there are five scores for each student.

```
int schoolScores[3][9][5];
```

One Dimensional Arrays

Hold five scores for a single student.

int studentScores[5];

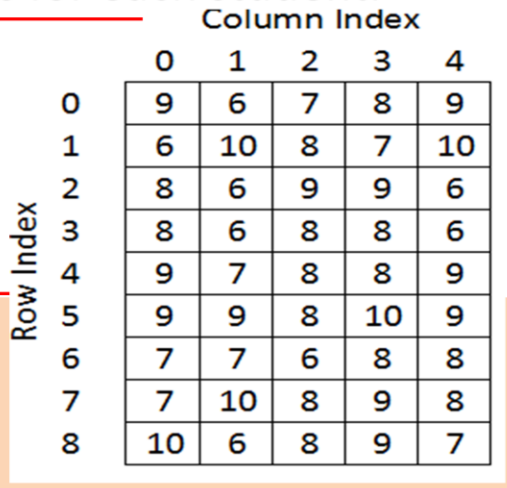


Starting with a single-dimensional array. I have int studentScores[5]; The data in this array that shows 7, 9, 7, 7, 7 is not initialized when the array is created. For this example, the data would be placed in the array by code when the program is being run.

Two Dimensional Arrays

Nine students, five scores for each student.

```
int classScores[9][5];
```



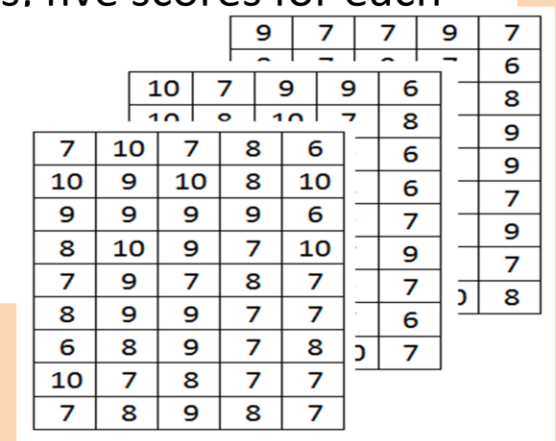
	0	1	2	3	4
0	9	6	7	8	9
1	6	10	8	7	10
2	8	6	9	9	6
3	8	6	8	8	6
4	9	7	8	8	9
5	9	9	8	10	9
6	7	7	6	8	8
7	7	10	8	9	8
8	10	6	8	9	7

Here is an example of creating a double-dimensional array for 9 students with 5 scores for each student. Again, the data in this array does not exist when the array is initially declared. Look closely at the square-brackets [ ] when the array is declared. The first set of square-brackets are used to define the number of rows, and a second set of square brackets define the number of columns in each row. This may be different if you are used to some other programming languages which would declare a double-dimensional array using parentheses and use a comma between the row and column definition. In Visual Basic, the array would be defined as Dim classScores(9, 5) As Integer

Three Dimensional Arrays

Three classes of nine students. five scores for each student.

int classScores[3][9][5];



Let's progress to defining a three-dimensional array. I might want to do this if I had three classes and 9 students in each class with 5 scores for each student. We are again using separate sets of square-brackets when defining the multi-dimensional array.

Parallel Arrays

```
int student1_Scores[5];
int student2_Scores[5];
int student3_Scores[5];
```

Another alternative to multi-dimensional arrays would be parallel arrays. Here I am declaring a separate array for each student. Although it is not too bad if there are only a few parallel arrays, it gets really messy as the number of parallel arrays grows.

Initializing a Two Dimensional Array

7	10	7	8	6
10	9	10	8	10
9	9	9	9	6

```
int classScores[3][5] = {
 {7, 10, 7, 8, 6},  {10, 9, 10, 8, 10},  {9, 9, 9, 9, 6}
};
```

Look closely at how the curly-braces and commas are used when initializing a double-dimensional array. There is an open and close curly-brace for the entire array, and then separate blocks of open/close curly-braces for each row. Commas separate the individual data for each row, and a comma is placed after the closing curly-brace for each internal row. The double-dimensional array is finished with a close curly-brace and a semicolon.

Initializing a Two Dimensional Array

```
int classScores[3][5] = { {7, 10, 7, 8, 6}, {10, 9, 10, 8, 10}, {9, 9, 9, 9, 6} };
```

```
int classScores[3][5] = {
   {7, 10, 7, 8, 6},
   {10, 9, 10, 8, 10},
   {9, 9, 9, 9, 6}
 };
```

7	10	7	8	6
10	9	10	8	10
9	9	9	9	6

The spaces or tabs when initializing data are optional. Just add spaces and tabs to make your code as readable as possible. You can place the data on one or more lines in your program for easier reading. Just remember where the curly-braces and commas are placed when using multiple lines and then use the same organization on fewer lines.

Nested **for** Loops

```
const int STUDENTS = 3;
const int QUIZZES= 5;
int classScores[ STUDENTS ][ QUIZZES ] = { {7, 10, 7, 8, 6}, {10, 9, 10, 8, 10}, {9, 9, 9, 9, 6} };
int classTotal = 0;
for (int student=0; student< STUDENTS; student++)
{
    for (int score=0; score< STUDENTS; score++)
        classTotal += classScores[student][score];
}
cout << "The class average is "  << classTotal/(STUDENTS* QUIZZES) << endl;
```

7	10	7	8	6
10	9	10	8	10
9	9	9	9	6

Here is a short code fragment that uses nested for loops to computes the average score for all the students in the class. This time, I am using constants to define the dimensions of the array. It is called a code fragment because it is missing the #include, main( ) and curly-braced for the block for main.

Before the outer loop starts, the classTotal is declared and initialized to zero. The outer for loop is going down through the rows for each student. The inner for loop is getting each quiz score for a student and adding it to the classTotal. When the inner for loop ends, the outer for loop moves to the next student and the inner for loop is run again and continues adding scores to the classTotal.

Finally, when all scores have been added into classTotal, the cout at the end displays "The class average is " and the average which is computed to be the classTotal divided by the number of scores for all students. The number of scores for all students is the product of STUDENTS times QUIZZES.

This concludes the introduction to arrays as used by C and C++. Bye for now.