# Lesson 8 - Arrays
# Contiguous Storage of Data

## Summary

In this lesson we describe how to create and use arrays of data with the aid of constant variables. We also explain how pointers and arrays can be used interchangeably.

### New Concepts

Arrays, constants, array-Pointer duality and casting.

## Arrays

In this section we are going to describe a useful C++ feature called Arrays. Frequently, programmers find themselves working with lists of data and arrays provide a convenient way to store that data. Before we begin, the important points to remember about arrays are (1) arrays hold a series of data containing the same type (`int`, `char` etc), (2) the size of an array must be provided when it is declared using a constant value and (3) "indexing" an array to retrieve an element begins with 0.

Without further ado let's see how to create an array to hold a series of three integer values. First we need a constant value for the size of the array. Take a look at line 8 of the `arrays.cpp` listing. Here we use the `const` keyword to declare a constant integer value, calling it `CONST_VALUE`. Using `const` tells the compiler that the value assigned to this variable cannot change throughout the duration of the program's execution. Because this is a `const` variable we must give it a value when we declare it, hence we set it to 3.

```
1 #include <iostream>
2
3 using namespace std;
4
5 float calc_avg(const int* arr, const int val);
6
7 int main(void) {
8    const int CONST_VALUE = 3;
```

arrays.cpp

On line 12 we declare our first array. We declare it to be an array of type `int` and give it the name `array_nums`. The square brackets [ ] after the name is referred to as the array index operator and contains the size of the array. The size must be a constant value so we use our `CONST_VALUE` variable for this purpose. Following the array index operator we immediately assign the elements of `array_nums` to some initial values within the curly brackets { }. Assigning initial values in this way is optional, as we shall soon see. On line 15 we use the array index operator again to access an element of the `array_nums` array. As mentioned earlier, array elements begin at the index 0, hence to display the first element of `array_nums` we must say `array_nums[0]`.

```
9    float ave;
10
11   // stack based 1-d arrays
12   int array_nums[CONST_VALUE] = {2, 4, 5};
13
```

```
14    // print out the first element
15    cout << "value at index 0 is " << array_nums[0] << "\n";
```

arrays.cpp

Arrays are not restricted to one dimension. For example, on line 17 we declare a two-dimensional array of `char` type. Note that 2-D arrays require double square brackets [ ][ ] to accommodate the extra dimension. We might use this 2-D array to represent the "board" for a game on naughts-and-crosses (or tic-tac-toe). Notice this time we use the "immediate" value 3 rather than our constant variable. Also, this time we did not initialise the array with values. On line 20 we set the middle square of our "board" to an 'X', hence we use the array index operator to access that element. Arrays of higher dimensions are possible but are a rare sight.

```
16    //stack based 2-d arrays
17    char naughts_n_xs[3][3];
18
19    // set the middle square to x
20    naughts_n_xs[1][1] = 'X';
```

arrays.cpp

The arrays so far have been declared on the stack, but we can also declare arrays on the heap. To declare an array on the heap, as usual, we use the `new` keyword. This returns a pointer, hence need a pointer to the type of array we wish to create (see line 22). The `new` keyword is followed by the type as usual, but we also tell the compiler how long the array should be within the array index operator. Now we have an array located on the heap, and we can use the pointer to access elements of the array, for instance when setting array elements to a value (see line 27).

This interchangeable use of pointers and arrays is a feature of C++. As a further example, note the call to the `calc_avg` function on line 30. We pass the `sum` array to this function to calculate an average value.

```
21    //arrays on the heap 1-d
22    int* sum = new int[CONST_VALUE];
23
24    for(int i = 0; i < CONST_VALUE; ++i)
25    {
26       cout << "enter a value...\n";
27       cin >> sum[i];
28    }
29
30    ave = calc_avg(sum, CONST_VALUE);
```

arrays.cpp

Examine the "calc_avg function" listing below. Note the first parameter is a `const int*` type called `arr` (line 1). This means that the first parameter to this function must be a constant pointer to an integer (i.e. a pointer whose value this function cannot change). But we passed the `sum` array when this function was called! Once again this demonstrates the interchangeable use of arrays and pointers. The name of the array `sum` is actually the same as referring to the memory address of the first element of the `sum` array. On line 5 however we see we can access elements of the `sum` array as usual. Note however that we must also tell the function how many elements are in the array and this is achieved by passing the constant `CONST_VALUE` as a parameter (line 1 of the calc_avg function listing).

This function simply iterates through the array, summing its values. Finally we return the average as a floating point type `float` by dividing this sum with the number of elements in the array. Note on line 7 the `(float)` statement before the `val` variable. This is called a "cast". We are casting the `val` variable from an integer to a floating point type. If we did not do this then we would be performing integer division and the result would be 2; the whole number without the remainder. As we want the floating point value therefore, we must "cast" `val` to a float.

```
1  float calc_avg(const int* arr, const int val) {
2     int sum = 0;
```

```
3
4     for(int i = 0; i < val; ++i)
5         sum += arr[i];
6
7     return (sum / (float)val);
8 }
```

calc_avg function

Finally, we return from the function and display the result of the `calc_avg` function on line 31. Note because we declared the `sum` array on the heap we should also release the memory to avoid a memory leak. We do this with the `delete` keyword as usual, however because `sum` is an array, we should follow the `delete` with the square brackets [ ] to denote that we are releasing an array, rather than a single value. Ensure that if you use the array form of new, that you always release it with the array form of delete.

```
31    cout << "the average is " << ave << "\n";
32
33    delete[] sum;
34
35    return 0;
36 }
```

arrays.cpp

**Under the Hood**

C++ defines arrays in row major order (rather than column major order). With this ordering the right most indices of arrays next to one another in memory. For example, the array values `arr[0][0]` and `arr[0][1]` are contiguous in memory and can typically be accessed very quickly. The code below shows both the correct and incorrect ways to access members of an array:

```
float testarr[500][500][500];
for(int i = 0; i < 500; i++) {
   for(int j = 0; j < 500; j++) {
      for(int k = 0; k < 500; k++) {
         testarr[i][j][k] = 0.0f; // good!
      }
   }
}
for(int k = 0; k < 500; k++) {
   for(int j = 0; j < 500; j++) {
      for(int i = 0; i < 500; i++) {
         testarr[i][j][k] = 0.0f; // bad!
      }
   }
}
```

arrays.cpp

In the first `for` loop, the array locations are mostly written to contiguously. At the level of the cache, this means that multiple array elements can be loaded from memory on each cache request and then accessed. With the second `for` loop this is not the case, but rather the next array element resides on a different 'cache line' and therefore must be retrieved from memory before it can be accessed. As these requests are costly in terms of execution time, this means the first `for` loop should require less execution time than the second.

## New Feature

New to C++11 is the range based for loop (sometimes referred to as a 'foreach' loop in other languages). Often we just want to perform some operation on every element of an array. In such situations we can use a shortened version of the standard for loop:

```
52    int arr[] = {1, 2, 3, 4, 5};
53    for(int& e : arr) { // new foreach syntax
54       e = e * e;
55    }
56    for(int i = 0; i < 5; i++) { // standard C++ for loop - ugly!
57       arr[i] = arr[i] * arr[i];
58    }
```

arrays.cpp

Lines 53 to 55 show the syntax for the range based for loop. The ampersand is used after the type (i.e. `int& e`) to indicate that we want the multiplication operation to affect the real values of the array rather than only copies.

## Exercises

1. Amend the code to create an array of characters on the stack, which contains your full name (including room for a space between your first and last names). Now create a function which prints to screen your name using the array you just created.

2. Create two arrays on the heap called "first" and "last". Copy your first name into the "first" array and your last name into the "last" array.

3. Create a "battleship" game using a 2-dimension array of booleans to represent the battle-ground. Use appropriate dimensions (say 5 by 5) and denote the presence of your "battleship" by setting the boolean values to `true` in array indices where your battleship is situated. Provide the user with a certain number of guesses to pick coordinates within the 2-D array. Your program should keep track of the number of guesses made and inform the user whether a guess is successful or not.