Arrays

Topics:

+ Array Declaration
+ Array Accession & Assignment
+ Array Initialization
+ Array Parameters
+ Array Miscellaneous

Resources:

- `array.cpp` - `misc.cpp`

Introduction

There are situations where you need to create multiple variables for a problem such as storing the names of people in an organization or the grades of an examination. Depending on the situation, you may need to make 10s to 100s of variables. Creating these variables, keeping track of their names, and manipulating them will be a frustrating, error prone and lengthy task. The solution to these problems is an *array*.

An array is a serial of same type objects.

`A Possible Int Array Declaration Of Size 9`

`r-values:`									
`l-values:`	`0x10`	`0x14`	`0x18`	`0x1C`	`0x20`	`0x24`	`0x28`	`0x2C`	`0x30`
`indices:`	0	1	2	3	4	5	6	7	8

An array is a sequential block of memory such that the addresses of each of its elements immediately follows each other with the size of each element equal to the size of the data type of the array. Furthermore, each element of an array can be referenced with the use of the name of the array and an *index* which is a nonnegative integer that relates to the position of the element in the array. A few important things to know about an array are

- All the elements of an array are of the same type.
- Only a single element of an array can be accessed (read from or written to) at a time besides during initialization.
- During an initialization of an array, some or all elements of the array can be assigned values; however, they must be initialized sequentially without gaps.
- When an array is a parameter of a function, it is always passed by reference; however, its syntax differs from the standard pass by reference parameter syntax as stated in the functions lecture.

Array Declaration

The declaration of an array is very similar to the declaration of a variable expect you attach the *subscript operator* [also known as the *indexer operator*] (`[]`) with a size between its symbols to the end of the name of the array as follows:

```
data-type identifier[size];
```

where *size* is either a positive int type literal or a constant int type variable assiged a positive value. **If *size* is not positive, you will receive an error [there is an expection for 0 for some compilers; however, initializing the size to 0 is not good programming practice]**. When you declare an array of size n, you are ideally declaring n variables all of the same type. For instance,

Example 1. Declare an int array of size 16 named *ages*.

Solution

```
int ages[16];
```

Example 2. Declare a double array of size 22 named *grades*.

Solution

```
double grades[22];
```

Example 3. Declare a string array of size 45 named *names*.

Solution

```
string names[45];
```

Last, multiple arrays can be declared at once along with variables such as below

```
int a, b[12], c, d[32];
```

The variables are *a* and *c*, and the arrays are *b* and *d*. All of them are int. Whenever multiple objects are declared at once, they are always the same type.

Array Accession & Assignment

As stated in the introduction, you can only access (read from or write to) a single element of an array at a time. To access an element of an array, you must use an index as follows:

```
array-name[index]
```

where *index* is a nonnegative integer that is strictly less than the size of the array. In fact, the index of an element is equal to one less than its position in the array. For instance, the index of the first element is 0, the second element is 1, the eighth element is 7, and so on. Thus, the possible indices of an array of size n are 0 through $n-1$. **Using an index outside that range will cause an error**.

Once you access an element of an array, it behaves like any other variable. So if you wsh to assign a value to an element of an array, you use the following syntax

```
array-name[index] = argument;
```

where *argument* must match the data type of the array. Furthermore, the elements of an array can be used as arguments wherever a variable of the same type can be used. For instance, Example 4. Assign the first three elements of *ages* the values 12, 36, and 73 respectively.

Solution

```
ages[0] = 12;
ages[1] = 36;
ages[2] = 73;
```

Example 5. Given that the double variable *avg* have been initialized, assign the last element of *grades* 3 more than twice *avg*.

Solution

```
grades[21] = 2 * avg + 3;
```

Example 6. Assign `"Joe"` to the third element of *names*; and then, assign the third element concatenated to itself with a hyphen between them to the nineteenth element of *names*.

Solution

```
names[2] = "Joe";
names[18] = names[2] +"-"+ names[2];
```

Array Initialization

Like a variable, an array can be initialized (declared and assigned a value in a single step). For an array, you can either initialize some of the elements of the array or all of them. Its syntax is

$$\left[const\right]^{?}\ \texttt{\textit{data-type identifier}}\,\texttt{[}\left[\texttt{\textit{size}}\right]^{?}\texttt{]}\ \texttt{=}\ \{\texttt{\textit{argument}}\left[,\texttt{\textit{argument}}\right]^{*}\};$$

When *size* is excluded, the size of the array is equal to the length of the initialization list and all its elements are initialized; **However, if *size* is excluded, the initialization list cannot be empty; otherwise, you may receive an error**. Whereas, when *size* is included, the size of the array is *size* and the first few elements of the array up to the length of the initialization list are initialized. **Furthermore, when *size* is included, the initialization list must consist of at most the *size* arguments; otherwise, you will receive an errror**. Examples of initializations are as follows

Example 7. Initialize a char array named *alphabet* of size 26 with the values `'a'`, `'b'` and `'c'`.

Solution

```
char alphabet[26] = {'a','b','c'};
```

Example 8. Initialize a double array named *scores* of size 5 with all zeroes.

Solution

```
double scores[] = {0.0,0.0,0.0,0.0,0.0};
```

Example 9. Initialize an int array named *count* of size 100 with the numbers 1 through 10.

Solution

```
int count[100] = {1,2,3,4,5,6,7,8,9,10};
```

Furthermore, it is important to note that you cannot place a gap in the initialization list. That is, you cannot just place a comma without an argument following it such as

```
double a[] = {1.0,,2.0};
```

The above example will cause an error. Likewise, if you initialize an array as constant, you can modify any elements later on. Typically, you will create constant array if its size is small and you know its values will not have to be changed.

Finally, char arrays can be initialized using a string literal. When a char array is initialized with a string literal, it ends with a null character; hence, it is known as a null terminated char array (or a *c-string*). Null terminated char arrays can be displayed by using the array name as seen in the following code segment

```
char a[5] = "hi", b[5] = {'h','i','\0'};

cout << a << "\n";
cout << b << "\n";
```

Normally, if you try to display an array, the address of its first element will be displayed. So to display all other arrays, you must display each element individually. Typically, you would define a function to display them.

Array Parameters

As stated in the functions lecture, arrays can be a parameter of a function. To recap, the syntax for an array parameter is

$$\left[\mathit{const}\right]^{?}\ \texttt{data-type identifier}\,\texttt{[]}$$

In the body of the function, the array parameter, as expected, is used as a local a normal array. However, it is important to reiterate, that arrays are always passed by reference, which means all modification made to the array will be reflected in the calling function. Furthermore, the elements of the array parameter may not been assigned values in the calling function. But if you do not want the array to be modified in the function, you need to make the parameter constant.

Likewise, to pass an array as an argument of a function call, you just use its name as you would a variable for a reference parameter argument. The code segment below shows a function definition that takes an array parameter and another function that calls it.

```
double sum(const double v[])
{
 return (v[0] + v[1] + v[2] + v[3] + v[4]);
}

int main()
{
 double a[8] = {1.2,33.2,21.5,7.1,9.0};
 cout << sum(a) << '\n';
 return 0;
}
```

The function `sum()` takes a constant double array parameter. This means, the elements of the parameter cannot be assigned new values as stated before. It returns the sum of the first five elements of the parameter. It is important to note, that a function does not have to manipulate or access all the elements of an array parameter. In fact, it does not know the size of the array parameter. This means you need to be careful when you use array arguments. **If the array argument has less elements than the function references, you will receive an error**. Typically, when an array is a parameter of function, the function will take an int parameter as well that represents the size of the array parameter. If you know which elements the function references of its array parameters, you do not need to worry able providing int parameters for the sizes of the array parameters. **Last, if you use a variable argument for an array parameter, you will receive an error**.

Array Miscellaneous

String objects (the string data type) can be manipulated as a char array (actually it is a char array and a couple of additional things). This means, you can modify or view the individual characters of the string with the subscript operator and an index. Since it is an object [a class object to be exact], it has *methods* (functions) that allow you access its *fields* (*data members* which will be discussed in the structures lecture) as well as manipulate them. The string object has a method `size()` [as well as the method `length()`] that returns the size of the string. Furthermore, the string object is null terminated; hence, you know you are at the end of the string when you reach the null character whenever you traverse the string. It is important to note that for any character array, char array or string object, the null character indicates the end of the string although there can be additional characters following the null character such as `"Hi\0All"`. The previous string will only be read as `"Hi"` (test it yourself). In the following code segment example, we will initialize a string object to a value, and then, change just its first and last characters, and display it.

```
string word = "bolt";
word[0] = 'p';
word[word.size()-1] = 'e';
cout << word << \n;
```

The above code will display `"pole"`.

Additionally, it is possible to make arrays of arrays called *multidimensional arrays*. Its declaration syntax is

data-type identifier[*size*][[*size*]]*;

For instance,

Example 10. Declare an int array of 5 arrays of size 16 named *gd*.

Solution `int gd[5][16];`

Example 11. Declare a double three-dimensional array named *point* that has the dimensions 8, 16 and 5 respectively.

Solution `double point[8][16][5];`

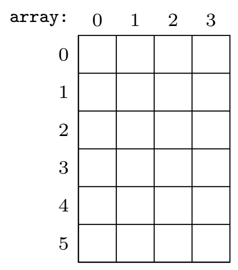
Example 12. Declare a bool five-dimensional array named *control* with the size of all dimensions equal to 4.

Solution `bool control[4][4][4][4][4];`

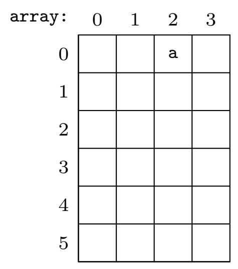
As a side note, you can also describe a two-dimensional array as a grid with rows (first size) and columns (second size). To access a single element of a multidimensional array, you need to use a valid index for each dimension of the array. Its syntax is

$$\texttt{\textit{array-name}[\textit{index}]}\,[\texttt{[\textit{index}]}]^{*}$$

where each *index* must be a nonnegative integer restrictly less than the size of its dimension. For instance, consider the two-dimensional array named *array* below



It has 6 rows and 4 columns (`array[6][4]`). Hence, valid indices for the first dimension are 0 through 5, and for the second dimension, they are 0 through 3. So, to assign a value to the element in the first row and third column, you would write `array[0][2] = a;` which will modify the array as follows



As with one-dimensional arrays, only a single element of a multidimensional array can be assigned a value at a time. However, you can access subarrays of multidimensional arrays be excluding trailing dimensions subscript operators. For instance, to access the third array of the two-dimensional array *array*, we would write `array[2]`. **It is important to note that you cannot have missing indices such as `array[][2];` it will produce an error**.

Furthermore, multidimensional arrays can be parameters of functions, however, all but the first dimension must have a fixed size. This means you can only pass multidimensional array arguments with the same trailing dimension sizes to the function rather than any multidimensional array with the same number of dimensions. For instance, consider the function defintion below

```
double DS(int a[][2])
{
 return a[0][1] + a[1][1];
}
```

The above function will **only** accept two-dimensional array arguments with a size of 2 for the second dimension. So, the first function call will produce an error while the second function call will not in the code segment below.

```
double a[2][3] = {{1,2,3},{3,6,9}}, b[3][2] = {{1,2},{3,4},{5,6}};

cout << DS(a) << "\n"; //error
cout << DS(b) << '\n';
```

Likewise, from the code segment above, you see that you can initialize a multidimensional array with an initialization list of initialization lists. Although you can use multidimensional array, you can use a one-dimensional array instead. To transition from a multidimensional array to a one dimensional array, you make the size of the one dimensional array equal to the product of the sizes of the dimensions of the multidimensional array. Next, the formula to transform a multidimensional index tuple to a one dimensional index is

$$f(i_1,\ldots,i_n) = \sum_{i=1}^{n-1} \prod_{j=i+1}^{n} size_j + i_n$$

where $size_i$ is the size of the ith dimension and the indices must be valid. For instance, for a two dimensional array with sizes m and n respectively, the one dimension formula will be

$$f(i_1, i_2) = n \times i_1 + i_2$$