Arrays

Topics:

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

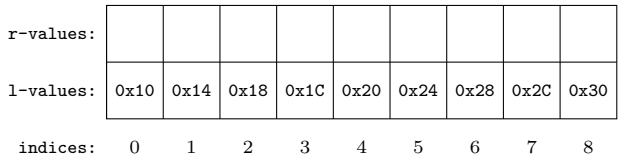
Resources:

- `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 orgainization or the grades of an examination. Depending of 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 this problem is to use an *array*.

An array is a serial of same type objects.



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 [the above illustration is of a possible int array declaration of size 9]. 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 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 that 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 instances,

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.

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 given that you start counting position from one. That is, 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 Idgrades 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

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

When *size* is excluded, the size of the array is equal to the length of the list and all its elements are initialized; **whenever you exclude *size*, make sure that the list is not empty unless you may receive an error**. Whereas, when *size* is included, the size of the array is *size* and the first few elements up to the length of the list are initialized. Furthermore, when *size* is included, the list must consists of at most the *size* arguments; otherwise, you will receive an errro. For instances,

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 initializer 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 an 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. Null terminated char arrays can be displayed by using the array name as in the following code segment

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

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

To display all other arrays, you must display each element individually. Typically, you would define a function to display an array.

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

$$[const]^{?}\ \textit{data-type identifier}\ []$$

In the body of the function, you use the array parameter the same way you would if it were standard local array. However, it is important to reiterate, that arrays are always passed by reference. All modification made to the array will be reflected in the calling function. If you do not want the array argument to be modified, you need to make the parameter constant; however, you cannot modify the array in the function as well. To pass an array as an argument of a function call, you just use its name. The code segment show 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[5] = {1.2,33.2,21.5,7.1,9.0};
 cout << sum(a) << '\n';
 return 0;
}
```