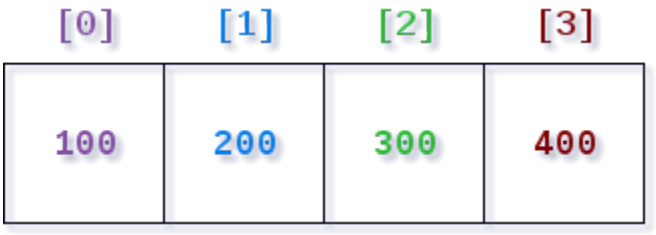
Array



An array is a collection of values of the same type (read **C-Strings** for **char** arrays). There are a few components of an array. Let's create an array of type **int** and dissect it:

```
int arrayVar[] = {100,200,300,400};
```

The first step is to specify the type of the array (**int** in this case).

The second step is to name your array just as you would during any variable declaration (**arrayVar** in this case).

The third step is to include square brackets **[]**. These denote that what you are creating is in fact an array and will be used when accessing a specific **element**. An **element** is an individual variable of the array. One thing to note here is if you leave the square brackets empty, the compiler will automatically determine the size so long as you initialize it at the same time. Otherwise, you would have to explicitly specify the size as the compiler needs to know how much memory to set aside, such as **double myArray[5];** which lets you initialize later. Keep in mind if you do this, each **element** will initially contain garbage values until properly initialized. Here in the example, the array is of size four as it is initialized with four separate integers. The following is also valid: **int arrayVar[4] = {100, 200, 300, 400};**

The optional fourth step is to initialize the array **{value0,…,valueN}**. Notice they are initialized within curly brackets and each value is separated by a comma (whitespace optional).

Here is an example program declaring and initializing an array as well as displaying its contents;

```
int main() {
    int array[3] = {1,2,3};
    cout << array[0] << ' ' << array[1] << ' ' << array[2] << endl;  }
```

Here, an array of type **int** with a size of 3 has been declared and initialized with each element containing **1 2 3** respectively. You can see that the beginning **element**, **array[0]** has the **index** of zero. In computer science, various things start with zero versus one. An **index** is the number within the square brackets and differentiates each **element**.

Additional Initialization Methods

The first additional method to initialize an array is by using a **for** loop and using **int i** as your index. For example:

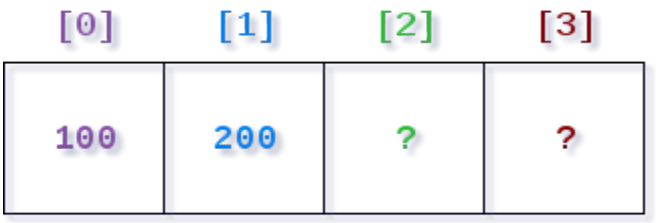
```
int myArray[5];
for(int i = 0; i < 5; i++)
    myArray[i] = 1; //Initializes elements 0 - 4 with 1
```

The second additional method is to set every element to zero in a terse way. This is done by simply putting a lone zero in curly brackets while declaring and initializing on the same line. Only the number zero behaves as such. Others initialize only first element.

```
int myArray[5] = {0}; //Initializes elements 0 - 4 with 0
```

The third additional method is to initialize only the first x amount of elements leaving the rest to be initialized later on. The remaining elements will contain garbage and should not be used until proper initialization. This leads you into partially filled arrays.

Partially Filled Arrays



Partially filled arrays are used when only some elements are known at compile-time, when elements are deleted from the array causing empty elements at the end (this method involves shifting elements to the front), or when the size of the array is a maximum boundary denoting the highest number of entries that you can enter, but may not necessarily need. With this method, it is important to keep track of the current number of used elements of the array so as to not access elements outside of the portion you want to use. An example of obtaining user input for a partially filled array is as follows:

```
int myArray[50]; //50 elements maximum
int used= 0; //Will be incremented after each entered element
int input; //Don't need to initialize as user will
cout << "Enter up to 50 tests for student. Use neg num to end \n";
for(int i = 0; i < 50; ++i) { //Remember < 50 as we start at element 0
    cin >> input;
    if(input < 0) //Grades cannot have negative numbers
        break; //One line statements do not need { }. Leave loop
    myArray[i] = input; //Put user input into array element i
    ++used; //Increase used for actual num of used elements.
}
```

Here, the user is able to enter up to 50 grades for a student. When they are finished, they enter a negative number, the loop terminates, and no more input is allowed. If 50 grades have been entered, the loop terminates as the variable **i** is now 50. Whatever the variable **used** is after the input is finished is the total number of valid elements of the array. Now, the variable **used** is able to be utilized when displaying results as a boundary or checking how many grades have been entered. Remember, any excess elements will contain garbage.

If an array is filled and the user needs to delete an element for whatever reason, all of the elements with a higher numbered index need to be shifted over one element. For example, if there is an array of size six and the element with index 4 needs deleted, then element with index 5 moves to index 4 and element with index 6 moves to index 5. The variable **size** needs to be decremented by one. Here is an example:

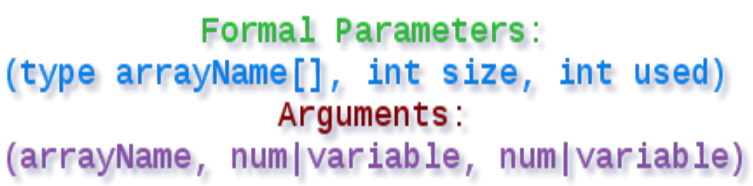
```
int main()
{
    int array[] = {1,2,3,4,5,6};
    int size = 6;

    cout << “Which index to delete?: \n”;
    int indexChoice;
    cin >> indexChoice;


    //indexChoice needs to be 0-5 inclusive to work
    if((indexChoice >= 0) && (indexChoice < size-1)) {
        for(int i = indexChoice; i < size-1; ++i) {
            array[i] = array[i+1]; } //Shift elements over
        --size; //Decrement after for loop is done
    } else if(indexChoice == size-1) { //Last element in array
        --size; //Decrement size by one. No shifting as last element
    }
}
```

In this program, a hardcoded **array** containing the numbers 1 - 6 inclusive and the variable **size** is present. The user is prompted which element to delete and gives their input. The **if** statement ensures that the index to be deleted exists. Remember the maximum index is always one less than the actual size. The **for** loop starts at the user's index choice and goes until **i** is one less than the last element. This is because in the body of the **for** loop the next element in sequence is accessed. During when **i** is 5, the final element at index 6 is also accessed. We cannot go over our bounds. The variable **size** is decremented after the loop and all appropriate elements are shifted. If the last element needs deleted, **size** simply needs decremented by one only as no shifting needs to take place.

Arrays in Functions



Passing arrays into functions requires up to three pieces of information. Focusing first on the **formal parameters**, the array variable is appended with the square brackets **[]**. This indicates that what is being passed is indeed an array. This does not tell the compiler what the size is. If you put a number inside the square brackets such as **arrayName[50]**, the compiler will ignore it. Therefore, you need a variable to be able to tell the size of the array. If the array is completely full, that will suffice. However, if you are dealing with a partially filled array, you need another variable to indicate how many elements have been properly filled. Onward to the **arguments**, the variable for the array does **not** include the square brackets. You are passing the variable only. The other two arguments, for the size and used variables in the function need to either be a literal number, such as 5 or a variable containing a number. An example focusing on the formal parameters and arguments is:

```
void function(double array[], int size, int used);
int main() {
    double doubleArray[3] = {'A', 'L'};
    int size = 3;
    int used = 2;
    function(doubleArray, size, used);
}
```

If you would like to pass a **single element** of an array to a function, it would look like this (using an integer array, for example):

```
void function(int arrayElement); //Formal parameter
function(intArray[5]);  //Function invocation and argument
```

The formal parameter and argument seem pretty much flipped from passing a whole array into a function. This is because **intArray[5]** is actually an integer number that is being held in the sixth element of the array. Remember that arrays being with zero. Therefore, the formal parameter must be of type **int** as you are passing only one element to the function.

Passing an array to a function will allow for manipulation of array as an array decays into a pointer to the first element. There is no need to pass-by-reference.