

# Array & String

In any programming language, we need a concept of list to store data items sequentially. *Array* is one of the most common and simple data structures available in C++ to do that. In this chapter, we are going to start by knowing how basic data is stored in memory with the concept of variables. Then we will explore different kinds of arrays with appropriate examples that illustrate several key behaviors of arrays. Finally, we will finish off with strings and their use in C++.

## 1.1 Data Storage Concept

Before we dive into learning the data structures used in any programming language, we need to know how data is stored in the memory of the computer. By having a clear understanding of that, we will be ready to venture further into the world of data structures seamlessly. Let us now start that journey by learning firstly, about variables.

### 1.1.1 Variables

It is well known that computers represent information using the binary number system. But we need to understand how we are going to represent information in our programming, which we will use as the basis for our computational processes. To do this, we need to decide on representations for numeric values and symbolic ones. We will use the concept of variables.

Let us think that, you are asked to retain the number 5 in your mental memory, and then you are asked to memorize also the number 2 at the same time. You have just stored two different values (5, 2) in your mental memory. Now, if you are asked to add 1 to the first number very first number, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Now subtract the second value from the first and you obtain 4 as a result. The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

1. `a = 5`
2. `b = 2`
3. `a = a + 1`
4. `result = a - b`

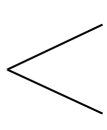
This is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them. Therefore, we can define a variable as a portion of memory to store a determined value. Each variable needs an identifier that distinguishes it from the others. For example, in the previous code, the variable identifiers were *a*, *b* and *result*, but we could have called the variables any names we wanted to, as long as they were valid identifiers.

So, a variable is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents. The identifier in computer source code can

be bound to a value during run time, and the value of the variable may thus change during program execution. In computing, a variable may be employed in a repetitive process: assigned a value in one place, then used elsewhere, then reassigned a new value and used again in the same way. Compilers have to replace variables' symbolic names with the actual locations of the data in the memory.

### 1.1.2 Memory Management

A *variable* is an area of *memory* that has been given a name. For example, if we look at the declaration `int x;`, it acquires four bytes of memory area and this area of memory that has been given the name `x`. One can use the name to specify where to store data. For example, `x=10;` is an instruction to store the data value 10 in the area of memory named `x`. The computer accesses its own memory not by using variable names but by using a memory map with each location of memory uniquely defined by a number, called the *address* of that memory location. To access the memory of a variable the program uses the `&` operator. For Example, `&x` returns the address of the variable `x`.

`int x;`  `&x` represents the memory area of variable named `x`.  
`x` represents the value stored inside the area of variable named `x`.

## 1.2 One-Dimensional Array

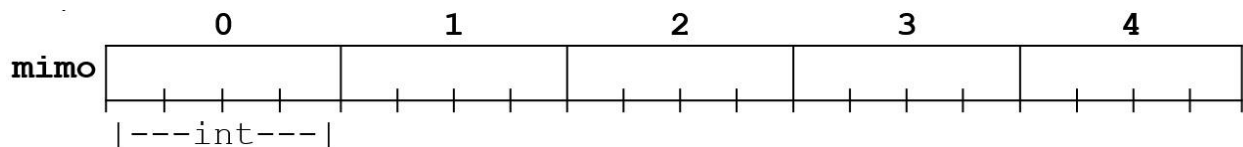
So far we have seen how a variable works and how the memory of the computer is associated with the use of a variable. We know that a variable can hold only one value of one type. A *variable* that contains one value of one type is a simple data structure. However, a lot of the time we are going to need to store a lot of values at the same time in a single variable. To do that, we need to know about another data structure called *Array*. There are mainly two kinds of arrays: One-Dimensional (1D), Multi-Dimensional (2D, 3D, 4D, etc.). We are going to be focusing on learning about 1D & 2D arrays in this chapter.

### 1.2.1 Definition & Structure

Let us assume you are asked to write a program where you need to store the CGPA of 1 student and output it. What approach will you take? Firstly, you will declare a float variable. Then take the CGPA of that 1 student as input and save it in that float variable. Then you are just going to use the print command in the programming language to print the variable's value. Pretty simple, right? Now, what if you are asked to store the CGPA of 200 students instead of only 1? One approach would be to declare 200 float variables and take 200 inputs separately for those 20 variables and write 200 print commands to print those 200 variables. As you can understand, this will mean the source code for this simple program will be huge and quite cumbersome. This is neither a good nor an efficient coding practice. This simple program can be written more shortly with the use of the right data structure. In this case, that data structure should be an array.

An array can hold a *series of elements* of the *same type* placed in contiguous memory locations. Each of these elements can be individually referenced by using an index to a unique

identifier. In other words, arrays are a convenient way of grouping a lot of values of the same type under a single variable name. For example, an array to contain 5 integer values of type `int` called `mimo` could be represented like this:



where each blank panel represents an element of the array, that, in this case, are integer values of type `int` with size 4 bytes. These elements are numbered/indexed from 0 to 4 since in arrays the first index is always 0, independently of its length.

### 1.2.2 Declaration

Now that we know the structure of an array, we need to know how to use it in a programming language. Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [total_number_of_elements];
```

where `type` is a valid data type (like `int`, `float` ...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array can contain. Therefore, to declare an array called `mimo` as the one shown in the above diagram it is as simple as:

```
int mimo [5];
```

The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a *constant integer value* since arrays are blocks of non-dynamic memory whose size must be determined before execution.

### 1.2.3 Initialization

When declaring a regular array (`int mimo[5];`) of local scope (within a function, for example) its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays are automatically initialized with their default values, zeros.

When we declare an array, we can assign initial values to each one of its elements by enclosing the values in braces `{ }` separated by commas. For example, `int mimo[5] = { 16, 2, 77, 40, 12071 };`. This declaration would have created an array like this:

	<code>mimo[0]</code>	<code>mimo[1]</code>	<code>mimo[2]</code>	<code>mimo[3]</code>	<code>mimo[4]</code>
<code>mimo</code>	16	2	77	40	12071

The number of values between braces `{ }` must not be larger than the number of elements that we declare for the array between square brackets `[]`.

An array can also be partially initialized. i.e. we assign values to some of the initial elements. For example, `int mimo[5] = { 16, 2};`. This declaration would have created an array like this:

	mimo[0]	mimo[1]	mimo[2]	mimo[3]	mimo[4]
mimo	16	2			
	--Uninitialized Elements--				

Here the first 2 values are assigned sequentially. The rest 3 elements are unassigned. Some more initializations:

```
float x[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
char vowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'}; is equivalent to string declaration: char vowel[6] = "aeiou";
```

## 1.2.4 Access

It is not helpful if after declaring and initializing an array, we cannot use the values inside it for the rest of the program. To use them, we first need to have access to them. Let us now take a look at how we can do that in this section of the chapter.

The number in the square brackets `[]` of the array is referred to as the '*index*' (*plural: indices*) or '*subscript*' of the array and it must be an integer number 0 to *one less than the declared number of elements*. We can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as: `name[index]`. For the declaration `int mimo[5];` the five (5) elements in `mimo` is referred in the program by writing: `mimo[0]`, `mimo[1]`, `mimo[2]`, `mimo[3]`, `mimo[4]`. Each of the array elements of `mimo` is an integer and each of them also acts as an integer variable. That is, whatever operation we can perform with an integer variable we can do the same with these array elements using the same set of rules.

Arrays have a natural partner in programs: the for loop. The for loop provides a simple way of counting through the numbers of an index in a controlled way. The for loop can be used to work on an array sequentially at any time during a program.

It is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays.

- one is to specify the size of arrays when they are declared - `char array[5];` Here the index (5) is always an integer constant.
- the second one is to specify indices for concrete array elements, like: `array[3] = '*';` Here the index (3) is always an integer or an integer expression.

Consider the following example:

```
char array[5];
array[7] = '*';
```

The assignment of `array[7]` is wrong as index 7 or element 8 in `array[5]` does not exist. But, C++ would happily try to write the character `*` at index 7. Unfortunately, this would probably be written in the memory taken up by some other variable or perhaps even by the operating system. The result would be one of the followings:

- The value in the incorrect memory location would be corrupted with unpredictable consequences.
- The value would corrupt the memory and crash the program completely! On Unix systems, this leads to a memory segmentation *fault*.

The second of these tend to be the result of operating systems with no proper memory protection. Writing over the bounds of an array is a common source of error. Remember that the array limits run from zero to the size of the array minus one.

Below is an access demonstration example of an array in C++:

```
1.  int mimo[5], a, b, i; // declaration of a new array
2.  mimo[2] = 75;         // store 75 in the third element of mimo
3.  a = mimo[2];          // copy/assign a with the third value of mimo
4.  cin >> mimo[2]); // read a value for the third element of mimo
5.  /* read 5 values for the five elements of mimo sequentially */
6.  for(i=0; i<5; i++)
7.      cin >> mimo[i];
8.  /* print 5 values for the five elements of mimo sequentially */
9.  for(i=0; i<5; i++)      ;
10.     cin >> mimo[i];
11. /* some more interesting accesses */
12. a = 4;
13. mimo[2] = a;
14. mimo[a] = 3;
15. b = mimo[a-2] + 2; //use of expression in index
16. mimo[mimo[a]] = mimo[2] + b;
```

### 1.2.5 Linear Search

Up to this point in the chapter, we should be quite clear about how an array works. In this section, we are going to take a look at one of the simplest searching techniques in computer science. It is called *Linear Search*.

A *Linear Search* is a technique of finding an element (or value) from a list (array) of elements. We can apply this technique to find an element within an array. For example, let us find `e` from an array called `lotsOfElements`. We begin searching `e` in `lotsOfElements` from its first value. We compare the first value with `e`. If `e` is equal to the first value, then we have found it. We do not need to search the rest of `lotsOfElements` anymore. However, if `e` is not equal to the first value of

lotsOfElements, we then move on to its second value. Now we check e with the second value. If they are not equal we move to the next one. We keep moving to the next value of lotsOfElements and compare it to e. Until we find a match, we keep doing like this. When we find a match, we stop and do not search the rest of lotsOfElements anymore. However, if we reach the end of lotsOfElements and still could not find the desired value, then the value is not found because it is not present in lotsOfElements.

We can implement *Linear Search* in C++ like below:

```
1.  int mimo[10] = {32,4,5,12,5,54,6,23,3,5}; // declaration of a new array
2.  int n;
3.  cout<<"Enter the number to be searched: "<<endl;
4.  cin>>n; // inputting the number to be searched in the array
5.  for(int i=0; i<10; i++){ // searching begins
6.      if (n == mimo[i]){
7.          break; // searching ends
8.      }
9.  }
10. cout<<n<<" was found in index "<<i<<" of the array."<<endl;
```

**OUTPUT: 5 was found in index 2 of the array.**

### 1.2.6 Insertion in Array

In the previous section, we have learned about the searching operation in an array in the form of *Linear Search*. One of the other important operations is *insertion* into an array. In this section, we are going to take a look at the insertion operation in three ways: inserting an element into an array from the back, front, and middle. For any kind of insertion into an array, we need to make sure the array has enough space for a new element to be inserted. An example illustrating these three kinds of insertions is given below in the form of a code snippet in C++ (with appropriate comments describing each line of code):

```
1.  int k, i, n=5, mimo[10]={2, 3, 5, 6, 7}; //partial initialization; n=total elements
2.  mimo[n++] = 8; // insert value 8 at the end of the array; increase n;
3.  /* insert value 1 at the beginning of array */
4.  for(i=n; i>0; i--) //shift all the values one index forward. i.e. the value
5.      mimo[i] = mimo[i-1]; //in index 1 goes to 2, 2 goes to 3,..., (n-1)th goes to nth.
6.  mimo[0] = 1; n++; //1 is inserted at index 1; n increases;
7.  // insert value 4 in the middle (index k=3) of the array
8.  k = 3;
9.  for(i=n; i>k; i--) //shift all the values one index forward. i.e. the value
10.      mimo[i] = mimo[i-1]; //in index k goes to k+1,..., (n-1)th goes to nth.
11.  mimo[k] = 4; n++; //4 is inserted at index k; n increases;
12.  for(i=0; i<n; i++) //printing all the values in the array after insert
13.      cout << mimo[i];
```

## 1.2.7 Deletion in Array

The *deletion* is another important operation performed in an array. Like *insertion* operation, it can also be done in three ways: deleting an element from the back, front, and middle. One thing to remember, we never actually delete anything from an array because there is no mechanism to free the space of an index of an array from the memory. Rather, we manipulate the array in such a way, so that when we print the array after the “deletion” operation, it prints the array without the “deleted” element. An example illustrating these three kinds of deletions is given below in the form of a code snippet in C++ (with appropriate comments describing each line of code):

```
1.  int k, i, n=8, mimo[10]={1, 2, 3, 4, 5, 6, 7, 8}; //n=total elements.
2.  n--; // Deleting the last element of the array. Decrease n; last element 8 is no
      longer part of list.
3.  /* delete value 1 from the beginning of array. */
4.  n--;          // deleting the value 1 will decrease the total elements n by one.
5.  for(i=0; i<n; i++) //shift all the values one index backward. The value in index
6.      mimo[i] = mimo[i+1]; //2 goes to 1, 3 goes to 2,..., nth goes to (n-1)th.
7.  k = 2; // delete value 4 from the middle (index k=2) of the array
8.  n--;          // deleting the value 4 will decrease the total elements n by one.
9.  for(i=k; i<n; i++) //shift all the values one index forward. i.e. the value
10.     mimo[i] = mimo[i+1]; //in index k+1 goes to k,..., nth goes to (n-1)th.
11. for(i=0; i<n; i++) //printing all the values in the array after insert
12.     cout<<mimo[i];
```

## 1.3 Two-Dimensional Array

We have learned about *one-dimensional (1D)* array previously and now we are going to learn about *multi-dimensional* array. To be more specific, we are going to learn about *two-dimensional (2D)* array. The basic ideas of the 1D array are still true for 2D array (sequence of elements of the same type) but the way we declare, initialize, and access a 2D array in the programming language is different than that of a 1D array. In this section, we are going to explore the 2D array in detail.

### 1.3.1 Definition, Structure & Declaration

Two-dimensional (2D) arrays can be described as "arrays of arrays". For example, a 2D array can be imagined as a Two-dimensional table made of elements of the same uniform data type.

		0	1	2	3	4
minu	0	minu[0][0]	minu[0][1]	minu[0][2]	minu[0][3]	minu[0][4]
	1	minu[1][0]	minu[1][1]	minu[1][2]	minu[1][3]	minu[1][4]
	2	minu[2][0]	minu[2][1]	minu[2][2]	minu[2][3]	minu[2][4]

In the figure above, minu represents a Two-dimensional array of 3 per 5 elements of type int. The way to declare this array in C++ would be: `int minu [3][5];`. The way to reference the 2nd element vertically and 4<sup>th</sup> horizontally or the (2 × 4) 8<sup>th</sup> element in an expression would be: `minu [1][3];`. Generally, for two-dimensional array, 1<sup>st</sup> dimension is considered as number of rows and the 2<sup>nd</sup> dimension is considered as number of columns. Here, we have 3 rows and 5 columns.

### 1.3.2 Initialization

Assigning values at the time of declaring a two-dimensional array can be any one of the following ways:

```
int minu[3][5] = {1,2,3,4,5,2,4,6,8,10,3,6,9,12,15};
int minu[3][5] = {{1,2,3,4,5},{2,4,6,8,10},{3,6,9,12,15}};
int minu[3][5] = {
    {1,2,3,4,5},
    {2,4,6,8,10},
    {3,6,9,12,15}
};
```

The internal braces are unnecessary but helps to distinguish the rows from the columns. Take care to include the semicolon at the end of the curly brace which closes the assignment. If there are not enough elements in the curly braces to account for every single element in an array, the remaining elements will be filled out with garbage/zeros. Static and global variables are always guaranteed to be initialized to zero anyway, whereas auto or local variables are guaranteed to be garbage.

### 1.3.3 Access

A nested loop is used to take input and give output. The input is taken in row (1<sup>st</sup> dimension) major. i.e. all the values of row 0 are scanned first, then the values of row 1, and values of row 2. For each row, value at column 0 is scanned first, then the value in column 1, and value in column 2. In the output, array a is used in row major. But array b is used in column (2nd dimension) major. i.e. all the values of column 0 are added first, then the values of column 1, and values of column 2. For each column, value at row 0 is added first, then the value at row 1, and value at row 2.

Consider the following example (the dark area at the end consists of the input and the output of this program; the yellow-colored text represents input given by the user and black colored text represents output):



```

1 // input & output of a 2D array
2 void main (void)
3 {
4     int i, j, a[3][3], b[3][3]={1,3,5,7,9,2,4,6,8};
5     for(i=0;i<3;i++)
6         for(j=0;j<3;j++)
7             cin>>a[i][j];
8     for(i=0;i<3;i++)
9         for(j=0;j<3;j++)
10            cout<<a[i][j] + b[j][i];
11 }

```

2 4 6 8 1 3 5 7 9  
3 11 10 11 10 9 10 9 17

## 1.4 Memory Access for Both 1D & 2D Array

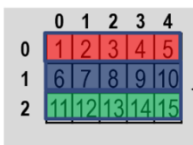
Two-dimensional arrays are just an abstraction for programmers since we can obtain the same results with a simple array just by putting a factor between its indices: `int minu [3][5]`; is equivalent to `int minu [15]`; ( $3 * 5 = 15$ ). Below we illustrate this similarity with a code snippet:

### ❑ 2D Array

```

int minu [3][5], H=3, W=5, n, m, i=0;
void main (void){
    for (n=0; n<H; n++)
        for (m=0; m<W; m++)
            minu [n][m]= ++i;
}

```



0	1	2	3	4
0	1	2	3	4
1	6	7	8	10
2	11	12	13	14

### ❑ 1D array

```

int minu [3 * 5], H=3, W=5, n, m, i=0;
void main (void){
    for (n=0; n<H; n++)
        for (m=0; m<W; m++)
            minu [ W * n + m ] = ++i;
}

```

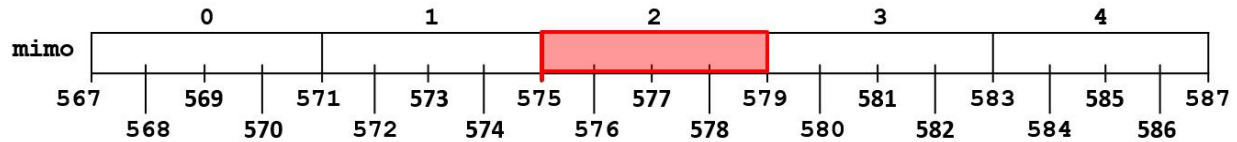


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

As memory is flat, in the above code for both 1D & 2D array the values are stored sequentially in the memory (just like the 1D array). The access for the 2D array, in that case, is just as the indexing of the array,

$$[(\text{Total\_column}) * (\text{row\_index}) + (\text{column\_index})]$$

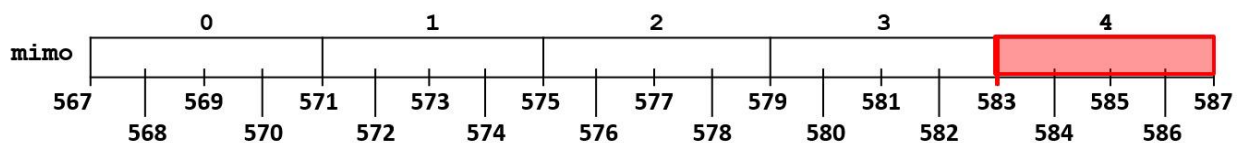
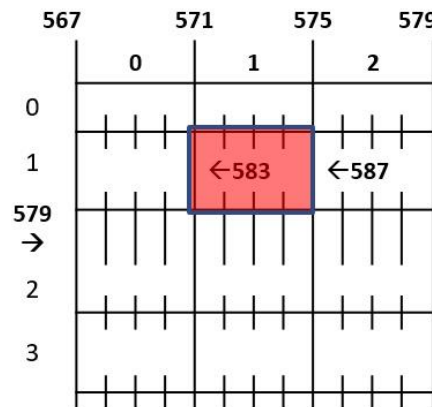
The memory of each element of an array can be accessed using the `&` operator.



$\&\text{mimo}[2]$  gives the memory location of the 3<sup>rd</sup> element of the array `mimo`. If the element is more than a byte, it gives the starting byte of the element. Let us consider the starting address of `int mimo[5]` is 567.  $\&\text{mimo}[2]$  will give us the memory location 575. `mimo[2]` will give us 4 bytes (`int`) of information starting from 575 to 579. The name of an array always refers to the starting location of the array. i.e. the first element of the array. So, `mimo` and  $\&\text{mimo}[0]$  both are the same. We can see the calculation for retrieving the memory address for  $\&\text{mimo}[2]$  below:

```
&array[index] = start_location_array + index * size_of_data
&mimo[ 2 ] = mimo (or &mimo[0]) + 2 * sizeof(int)
&mimo[ 2 ] = 567 + 2 * 4 = 575
```

Consider a 2D array `mimo[R][C]` each element addressed by  $\&\text{mimo}[i][j]$ , where **R** = total element in 1<sup>st</sup> dimension, **C** = total element in 2<sup>nd</sup> dimension,  $0 \leq i < R, 0 \leq j < C$ . Let `int mimo[4][3]`; Here, `mimo` or  $\&\text{mimo}[0][0]$  gives us the starting memory location 567. `mimo[1][1]` will give us 4 bytes (`int`) of information starting from 583 to 587.



```
&array[i][j] = start_location + (i * (C * size_of_data)) + (j * size_of_data)
&mimo[1][1] = mimo + (1 * (3 * sizeof(int))) + (1 * sizeof(int))
&mimo[1][1] = 567 + (1 * 3 * 4) + (1 * 4) = 583
```

There is a general way to access the memory location of a 2-dimensional array. For an array `int mimo[R][C]`; and  $0 \leq i < R, 0 \leq j < C$ :

- `mimo[i] = &mimo[i][0]` represents the starting address of *i*th row.
- `mimo[i]` skips *i* number of rows each with **C** number of elements from the `start_location` of the array.
- So, `mimo[i] = start_location + (i * C elements)`, where **C** elements are counted in bytes based on the `size_of_data`, here `int`.
- So, `mimo[i] = start_location + (i * (C * size_of_data))`.
- So, `mimo[i] = mimo(or &mimo[0][0]) + (i * (C * sizeof(int)))`.

## 1.5 String

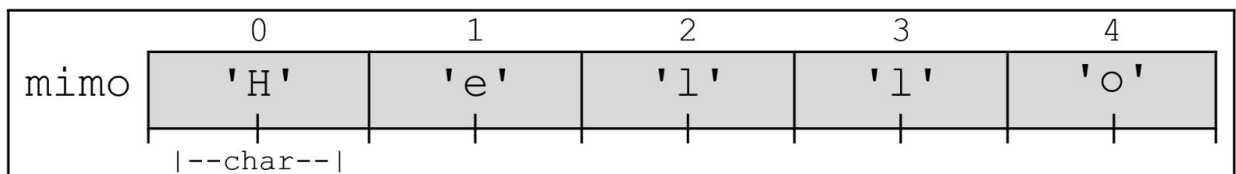
After learning arrays in detail, now it is time for us to know about one of the most important data structures in any programming language: *string*. In this section, we are going to learn about it in detail.

### 1.5.1 Definition & Structure

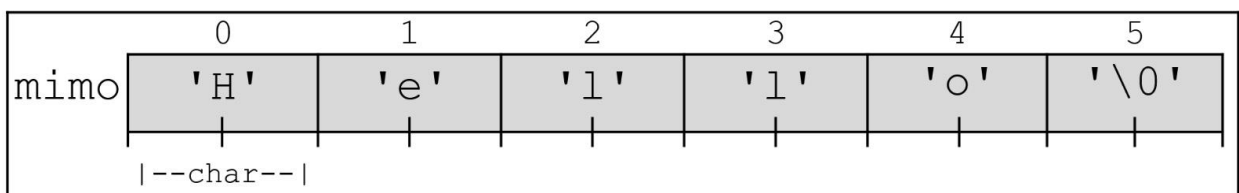
Strings are a sequence of characters representing a piece of text. In programming languages a string is represented as some characters enclosed by double quotes: "This is a string". A string is mainly declared using an array of characters. The main difference between *a simple array of characters* and *an array of characters representing string* is the end marker given at the end of a string. The standard library functions can recognize this end marker as being the end of the string. The end marker is called the zero (or NULL) byte because it is just a byte which contains the value zero: `\0`. Programs rarely get to see this end marker as most functions that handle strings use it or add it automatically.

### 1.5.2 Declaration & Initialization

Declaration of a string is just an array of characters: `char mimo [5];`. But things are different when we initialize during declaration. An array to contain 5 character values ('H', 'e', 'l', 'l', 'o') of type `char` called `mimo` could be represented like this: `char mimo [5]={ 'H', 'e', 'l', 'l', 'o'};`.



But, to represent the same text as a string in C++ double quotation (") is used to bound the text. And, a NULL character is added at the end of the text. So, because of this NULL, we need to declare an additional slot in the array. `char mimo [6]= "Hello";`. The standard library functions can recognize NULL (`\0`) as being the end of the string.



### 1.5.3 Access, Input, Output

Consider the following example (the dark area at the end is the output of this program; the red-colored text represents input given by the user):

```
1 // null-terminated sequences of characters
2 void main (void)
3 {
4     char Question[]="Please, enter first name: ";
5     char Greeting[] = "Hello";
6     char FirstName[80];
7     cout<<Question;
8     cin>>FirstName;
9     cout<<Greeting<<" ";<<FirstName);
10 }
```

Please, enter first name: **John**

Output: **Hello, John!**

Lines 4-6 declare three arrays of characters. The first two are initialized as strings with a NULL character at their end. Line 7 outputs the string in Question. The same can be found in line 8, where we have cin with the array FirstName. The address indicates the location in the memory from where the processing will start. The NULL character indicates where the processing will stop for cout. And for cin, after the processing (i.e., after the string input) a NULL character is automatically added at the end of the string.

Instead of asking for the first name only, if we would have asked for the whole name, this program might not work properly. That is if we would have given input John Rambo instead of only John the output will be as follows –

Please, enter your first name: **John Rambo**  
Hello, John!

Still, it shows John after Hello, not John Rambo. As we know, that cin always stops at white spaces during taking input. So, after taking John as input cin receives a space indicating the end of input. So it never even goes for Rambo. To overcome this, there is another function cin.get() which takes only 2 parameters: *variable name of the string* and *its size* (maximum size of the character array). So, just writing cin.get(FirstName,80); instead of cout<<Question; will give the following output –

Please, enter your first name: **John Rambo**  
Hello, John Rambo!

### 1.5.4 String Handling Functions

Strings are often needed to be manipulated by a programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. C++ supports a large number of string handling functions. There are numerous functions defined in the "cstring" header file (library). For all such library functions, a NULL character is assumed to be at the end of the existing string and a NULL character is always included at the end of the new/updated/changed string. Few commonly used string handling functions are discussed below:

Function	Work Of Function
<code>strlen(str)</code>	Calculates the length of string <code>str</code> ; the total number of characters from the given starting address by <code>str</code> until a NULL encounters.
<code>strcpy(s1, s2)</code>	Copies a string <code>s2</code> to another string <code>s1</code> ; all the characters starting from the given starting address by <code>s2</code> until a NULL encounters, will be copied in sequentially from the given address by <code>s1</code> .
<code>strcat(s1, s2)</code>	Concatenates (joins) two strings; all the characters starting from the given starting address by <code>s2</code> until a NULL encounters, will be copied in sequentially from the NULL character at the end of <code>s1</code> .
<code>strcmp(s1, s2)</code>	Compares two strings; if <code>s1</code> and <code>s2</code> are equal <code>strcmp</code> returns 0, if <code>s1</code> is alphabetically (compares ASCII value) lower than <code>s2</code> then <code>strcmp</code> returns <0 and otherwise returns >0;

### 1.5.5 String as Object

In C++, there is another way to store a string in a variable, by using a string object. Unlike character arrays, there is no fixed length for string objects. It can be used as per the requirement of the programmer. Unlike a string that is a character array, a string object does not have a NULL character at the end of it.

Instead of using `cin` or `cin.get()` functions for accepting strings with white spaces as input, we can use `getline()` function. The `getline()` function takes the input stream as the first parameter which is `cin` and string object as the second parameter.

The "string" header file contains library functions that can be used on string objects for manipulation. [See *Reference* at the end of the chapter for more info on that]. To use library functions of "cstring" on a string object, it needs to be converted to a string of character array with a NULL character at the end. There is a `length()` function that can get the total size of the string object. Run the following example in the IDE to see what it outputs,

```

#include<iostream>
using namespace std;

int main() {
    string s;
    getline(cin,s);
    cout<<"String object accessed using array and loop...\n";
    for(int i=0; i<s.length(); i++){
        cout<<s[i];
    }
    cout<<endl;
    cout<<"Not using arrays or loops\n";
    cout<<s;

    return 0;
}

```

## 1.6 Exercises

1. Explain the differences between 1D & 2D arrays with appropriate examples.
2. What is the significance of NULL in a non-object string? Motivate your answer with an example.
3. What is the limitation of cin when printing a string in C++? What is the alternative to overcome such limitations? Explain with appropriate examples.
4. Write a C++ code to print the following array in reverse order:  
int y[10] = {32,4,1,2,5,5,23};
5. If you have an array, int x[4] = {8,5,3,6}, what will be the output of the following statements:  
cout<<x[4];  
cout<<x[1];
6. Given a 2D array, int test[3][2] = {{21,25},{32,41},{35,61}} write a loop in order to print the following from test:  
25, 41, 61
7. What will you change in the declaration/ initialization of the 2D array test (in question 6) in order to make it a 1D array?
8. Again, take the array test (in question 6). Suppose, the starting address of the array is 1046. What will be the address of 41? What will be the address of 41 when you have converted test to a 1D array (in question 7)?
9. Write code in C++ to implement the functions: *strlen*, *strcmp*, *strcat*, *strcpy* by yourself.

## 1.7 References

1. [https://en.wikipedia.org/wiki/Array\\_data\\_structure](https://en.wikipedia.org/wiki/Array_data_structure)
2. [https://en.wikipedia.org/wiki/Array\\_data\\_structure](https://en.wikipedia.org/wiki/Array_data_structure)
3. <https://www.programiz.com/cpp-programming/strings>
4. <https://cal-linux.com/tutorials/strings.html>

5. <http://www.cplusplus.com/reference/cstring/>
6. <http://www.cplusplus.com/reference/string/string/>
7. <https://cal-linux.com/tutorials/strings.html>