9. Arrays and Strings

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

9.1 Declaration of arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

9.2 Initialization of array

You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

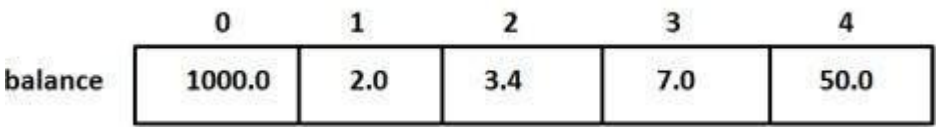
The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array: If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above:



9.3 Accessing elements of array

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable.

9.4 I/O of arrays

Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream.h>
#include<conio.h>

int main ()
{
   int n[ 10 ];

   for ( int i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100;
   }
   cout << "Element" << “    “ << "Value" << endl;

   for ( int j = 0; j < 10; j++ )
   {
      cout << “   “<< j << “    “ << n[ j ] << endl;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

9.4.1 To reverse the elements of an array

The array elements are in a particular order and they can be printed by using loops as follows in reverse order as:

```
#include <iostream.h>
#include<conio.h>

void main ()
{
   int n[10] = {1,2,3,4,5,6,7,8,9,10};

   for (int i=9;i>=0;i--)
   {
      cout << n[i];
   }
}
```

The above code will print the array elements in reverse order.

The code to reverse the array elements is as follows:

```
#include <iostream.h>
#include<conio.h>

void main ()
{
   int n[10] = {1,2,3,4,5,6,7,8,9,10};
   int temp[10];

   int j=0;
   for (int i=9;i>=0;i--)
   {
      temp[i] = a[j];
      j++;
   }

   for(i=0;i<10;i++)
   {
      a[i] = temp[i];
   }
}
```

9.5 Passing arrays as arguments to a function

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three

declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

Formal parameters as a pointer as follows:

```
void myFunction(int *param)
{
...
}
```

Way-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
{
...
}
```

Way-3

Formal parameters as an unsized array as follows:

```
void myFunction(int param[])
{
...
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
double getAverage(int arr[], int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
   }

  avg = double(sum) / size;

  return avg;
}
```

Now, let us call the above function as follows:

```
#include <iostream.h>
#include<conio.h>

// function declaration:
double getAverage(int arr[], int size);

int main ()
{
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.4
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

9.6 Multidimensional arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C++ data type and arrayName will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

```
#include <iostream.h>
#include<conio.h>

int main ()
{
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

   // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ )
```

```
      {
         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

C++ program to read and display an entire line entered by user.

```
#include <iostream.h>
#include<conio.h>

int main() {
    char str[100];
    cout<<"Enter a string: ";
    cin.get(str, 100);
    cout<<"You entered: "<<str<<endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, cin.get function can be used. This function takes two arguments. First argument is the name of the string(address of first element of string) and second argument is the maximum size of the array.

9.7 String as array of characters, Initializing string variables and I/O of strings

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String:

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

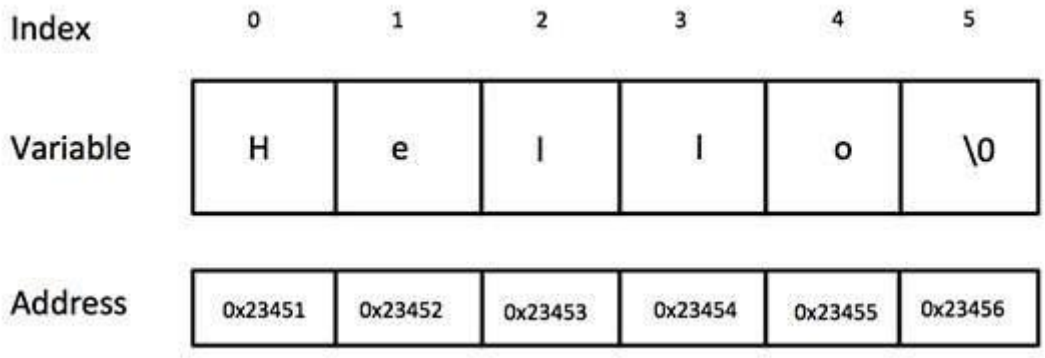
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream.h>
#include<conio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   cout << "Greeting message: ";
   cout << greeting << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Greeting message: Hello
```

The String Class in C++:

The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality. We will study this class in C++ Standard Library but for now let us check following example:

At this point, you may not understand this example because so far we have not discussed Classes and Objects. So can have a look and proceed until you have understanding on Object Oriented Concepts.

```
#include <iostream.h>
#include <string.h>
#include<conio.h>

int main ()
{
   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total lenghth of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

9.8 String manipulation functions (strlen, strcat, strcpy, strcmp)

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function	Purpose
1	**strcpy(s1, s2);**	Copies string s2 into string s1.
2	**strcat(s1, s2);**	Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**	Returns the length of string s1.
4	**strcmp(s1, s2);**	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**	Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**	Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <iostream.h>
#include <cstring>
#include<conio.h>

void main ()
{
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

9.9 Passing strings to a function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream.h>
#include<conio.h>
void display(char s[]);

void main() {
    char str[100];
    cout<<"Enter a string: ";
    cin.get(str, 100);
    display(str);
}

void display(char s[]) {
    cout<<"You entered: "<<s;
}
```

Output

```
Enter a string: Programming is fun.

You entered: Programming is fun.
```