Chapter Table of Contents

Chapter 1.2

Arrays, String and Structures

Aim

To help the students understand fundamentals of Arrays, String, Structures and Unions

Instructional Objectives

After completing this chapter, you should be able to:

- Explain the fundamentals of Array
- Analyse different methods to accept strings
- Discuss how to define and declare a structures and unions
- Illustrate the declaration and definition of Enumerated data type

Learning Outcomes

At the end of this chapter, you are expected to:

- Design and execute programs to demonstrate single dimensional array and multi-dimensional array
- Identify different string manipulation functions
- Design and implement a program to demonstrate structures and unions
- Differentiate structure and union with example
- Explain enumerated data types with example

1.2.1 Introduction

In computer languages, we need to group together data items of the same type. The most basic mechanism that accomplishes this in C++ is the array. Arrays can hold a few data items or tens of thousands. The data items grouped in an array can be simple types such as int or float, or they can be user-defined types such as structures and objects.

Arrays exist in almost every computer language. Arrays in C++ are similar to those in other languages and identical to those in C. In this chapter we will look first at arrays of basic data types such as int and char. Then we will examine arrays used as data members in classes and arrays used to hold objects. Thus this Chapter is intended not only to introduce arrays, but to increase your understanding of object oriented programming.

In Standard C++, the array is not the only way to group elements of the same type. A vector, which is part of the Standard Template library, is another approach.

Further, in this chapter, we will also look at two different approaches to strings, which are used to store and manipulate text. The first kind of string is an array of type char and the second is a member of the Standard C++ string class.

1.2.2 Array Fundamentals

In programming, we have different data types which can be used to store different kind of data. But in some programs, we need to handle similar types of data.

There may be a need to store marks of more than one student depending upon the input from user or to store salaries of more than one employee. This can be handled in C++ programming using arrays.

An array is used to process a collection of data of the same data type. An array is a collection of similar data elements or fixed number of elements of the same type stored sequentially in memory. There are two types of array.

1. Single dimensional array
2. Multi-dimensional array

(i) Single Dimensional Arrays

A single/one-dimensional array is a list of related variables. The general form of a one-dimensional array declaration and syntax are shown below.

Syntax:

```
type arrayname [size];
```

- **type:** base type of the array, determines the data type of each element in the array.
- **size:** how many elements the array will hold (dimension).
- **arrayname:** the name of the array.

For examples,

```
int age[4];
float Salary[100];
char name[40];
```

To declare an integer array named age of four elements, we write

```
int age[4]
```

The elements of an array can be accessed by using an index into the array. Arrays in C++ are zero-indexed, so the first element has an index of 0. So, to access the second element in age, we write age[1] and to access the fourth element in age, we write age[3] and so on ; The value returned can then be used just like any other integer. Like normal variables, the elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program.

The memory allocation for the array elements is shown in the Figure 1.2.1

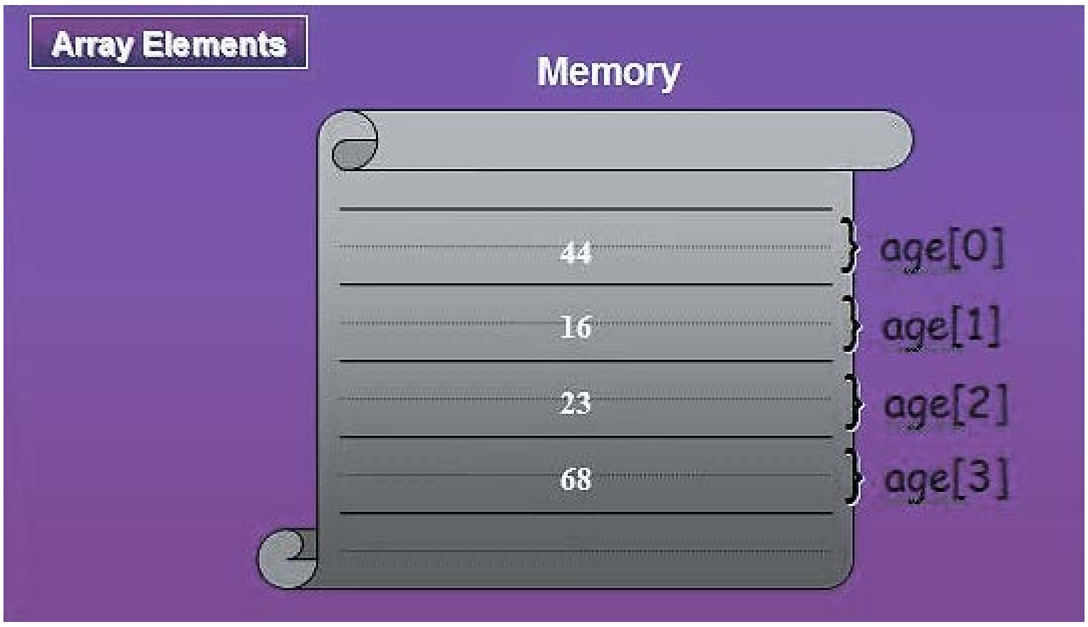


Figure 1.2.1: Memory Allocation for Array Elements

There are several ways to initialize the array.

One way is to declare the array and then initialize some or all of the elements one by one:

```
int age[4];
age[0] = 44;
age[1] = 16;
age[2] = 23;
age[3] = 68;
```

Another way is to initialize some or all of the values at the time of declaration:

```
int age[4] = {34, 15, 18, 19};
```

The initialization of an array is depicted in the Figure 1.2.1.

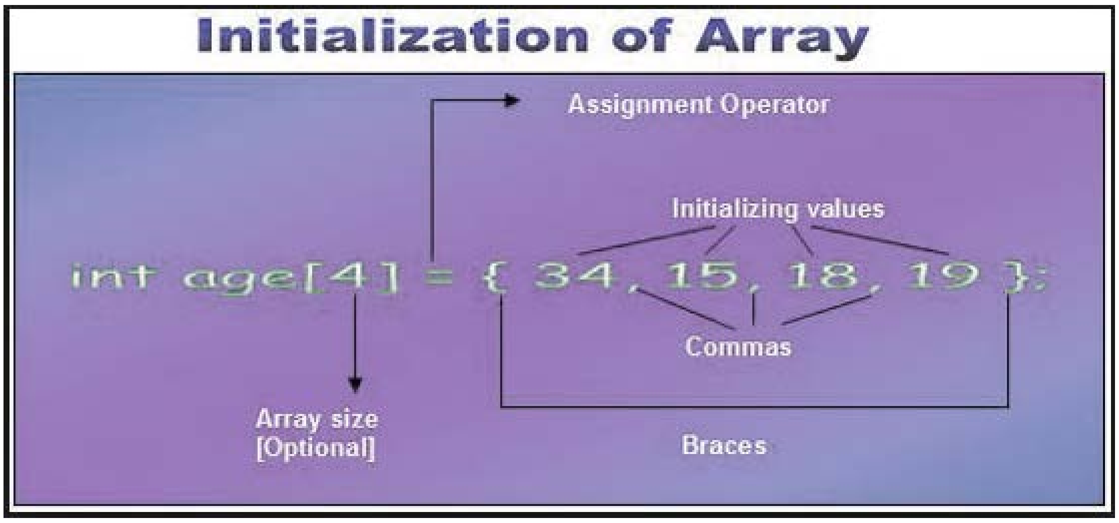


Figure 1.2.2: Initialization of Array

Sometimes it is more convenient to leave out the size of the array and let the compiler determine the array's size for us, based on how many elements we give it:

```
   int age[] = { 44,16,23,68,21, 23, 18, 19 };
```

Here, the compiler will create an integer array of dimension 8. The array can also be initialized with values that are not known beforehand:

Example program

```
#include<iostream.h>
using namespace std;
int main()
{
   int age[4];
   cout<< “Please enter age of 4 persons“ <<endl;
   for(int i = 0; i < 4; i++)
   cin>> age[i];
   cout<< “Entered ages are:“;
   for(int i = 0; i < 4; i++)
   cout<< “ “ << age[i];
   cout<<endl;
}
return 0;
```

Note that when accessing an array the index given must be a positive integer from 0 to n-1, where n is the dimension of the array. The index itself may be directly provided, derived from a variable, or computed from an expression:

```
   age[3];
   age[i];
   age[i+3];
```

Arrays can also be passed as arguments to functions. When declaring the function, simply specify the array as a parameter, without a dimension. The array can then be used as normal within the function.

For example,

```
#include<iostream.c>
using namespace std;
int sum(constint array[], constint length)
{
   long sum = 0;
   for(int i = 0; i < length; sum += array[i++]);
   return sum;
}
int main()
{
   intarr[] = {1, 2, 3, 4, 5, 6, 7};
   cout<< "Sum: " << sum(arr, 7) <<endl;
   return 0;
}
```

The function sum takes a constant integer array and a constant integer length as its arguments and adds up length elements in the array. It then returns the sum and the program prints out Sum: 28. It is important to note that arrays are passed by reference and so any changes made to the array within the function will be observed in the calling scope.

Character Array Initialization

Character arrays that will hold strings allow a shorthand initialization that takes this form:

```
   char array-name[size] = “string”;
```

For example, the following code fragment initializes str to the phrase “hello”:

```
   charstr[6] = “hello”;
```

This is the same as writing char str[6] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

Remember that one has to make sure to make the array long enough to include the null terminator.

(ii) Multi-dimensional Arrays

C++ also supports the creation of multidimensional arrays, through the addition of more than one set of brackets. Thus, a two-dimensional array may be created by the following:

```
   typearrayName[dimension1][dimension2];
```

The array will have dimension1 x dimension2 elements of the same type and can be thought of as an array of arrays. The first index indicates which of dimension1 subarrays to access and then the second index accesses one of dimension2 elements within that subarray. Initialization and access thus work similarly to the one-dimensional case:

```
#include<iostream.h>
using namespace std;
int main()
{
   intarr[2][4];
   arr[0][0] = 6;
   arr[0][1] = 0;
   arr [0][2] = 9;
   arr[0][3] = 6;
   arr[1][0] = 2;
   arr [1][1] = 0;
   arr [1][2] = 1;
   arr[1][3] = 1;
   for(int i = 0; i < 2; i++)
   for(int j = 0; j < 4; j++)
   cout<<arr[i][j];
   cout<<endl;
   return 0;
}
```

The array can also be initialized at declaration in the following ways:

```
   intarr[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };
```

intarr[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } }; Note that dimensions must always be provided when initializing multidimensional arrays, as it is otherwise impossible for the compiler to determine what the intended element partitioning is. For the same reason, when multidimensional arrays are specified as arguments to functions, all dimensions but the first must be provided (the first dimension is optional), as in the following:

```
   intaFunction(intarr[][4]) { … }
```

Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory. Declaring intarr[2][4]; is the same thing as declaring intarr[8];.

2 Dimensional arrays are used to represent matrices.

Multidimensional arrays can be used when you want to store values in a tabular format in rows and columns. ***For example,*** a railway timetable.

A Multidimensional array can be used to input data or to perform operations on the data set. ***For example,*** if you want to store 10 employee names and their respective salaries.

We can create a 3-D array for storing height, width and length of each room on each floor of a building.

Differences between Single and Multidimensional arrays:

Single dimensional arrays can be used to store single list of elements of similar data types whereas multidimensional arrays stores a 'list of lists' or 'array of arrays' or 'array of one dimensional arrays'.

For declaring a one dimensional array we need to specify a single dimension whereas while declaring a multidimensional array we need to specify multiple dimensions.

For example, single dimensional array can be used to store a list of marks of one student whereas a 2 dimensional array can be used to store list of marks of n number of students in a matrix form.

Self-assessment Questions

1) An array is a ____________.
    a) Collection of similar data elements
    b) Collection of different data elements
    c) Collection of data
    d) Collection of files

2) Every array element is accessed using a
    a) Index b) Name
    c) size d) Name and size

3) If an array is declared as intarr[5][5], how many elements can it store?
    a) 5 b) 10
    c) 25 d) 0

1.2.3 String Fundamentals

In C++, strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate '#include' directive, as shown below:

```
#include <string>
using namespace std;    // Or using std::string;
```

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";
```

Figure 1.2.3 shows the memory presentation of above defined string in C/C++:

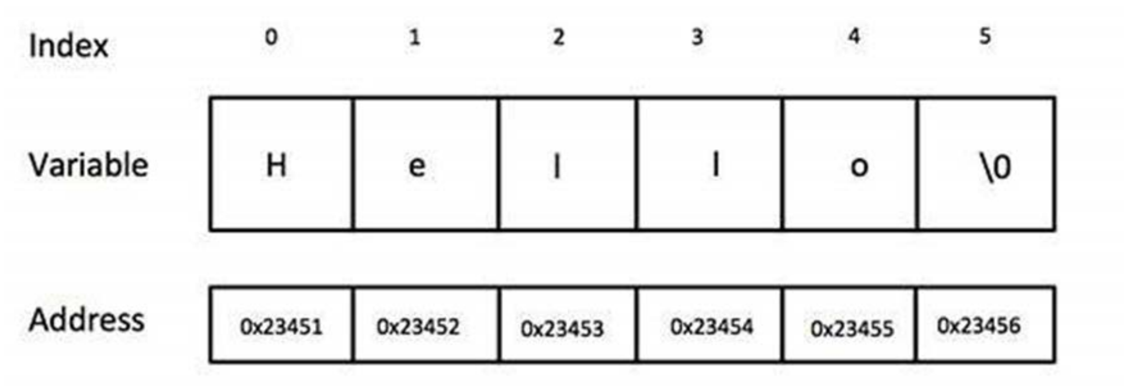
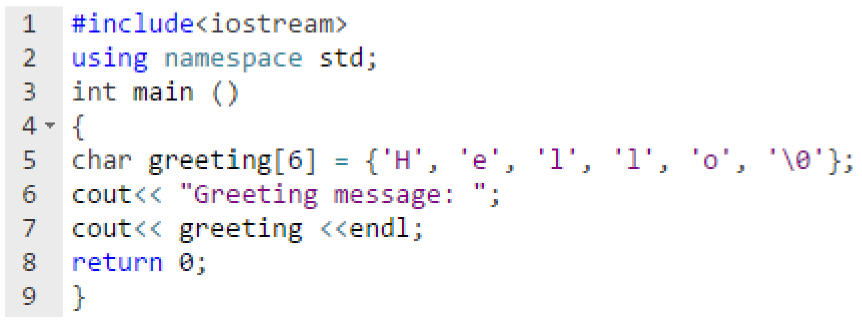


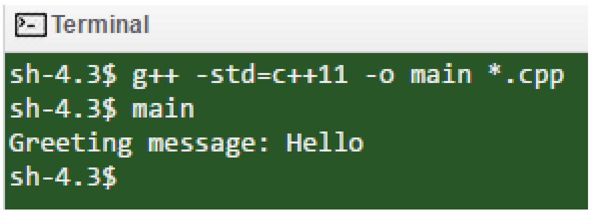
Figure 1.2.3: Memory Presentation of String

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>
using namespace std;
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 like something as follows:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Greeting message: Hello
sh-4.3$
```

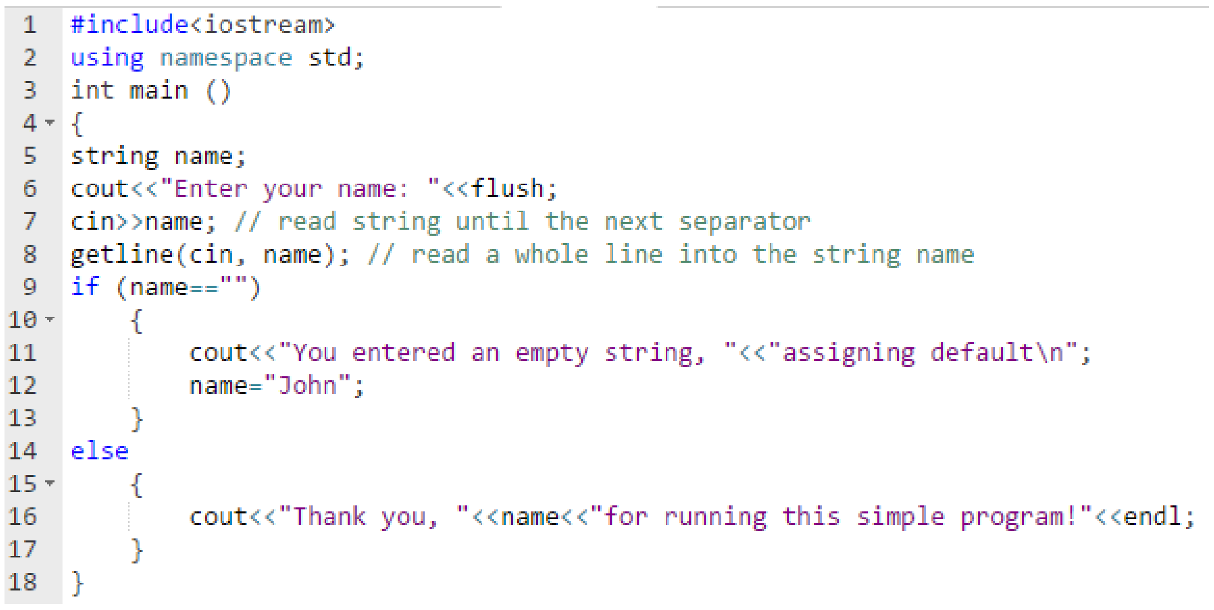
We now use **string** in a similar way as built-in data types, as shown in the example below, declaring a variable **name**:

```
String name;
```

Unlike built-in data types (**int, double**, etc.), when we declare a **string** variable without initialization (as in the example above), we do have the guarantee that the variable will be initialized to an empty string - a string containing zero characters.

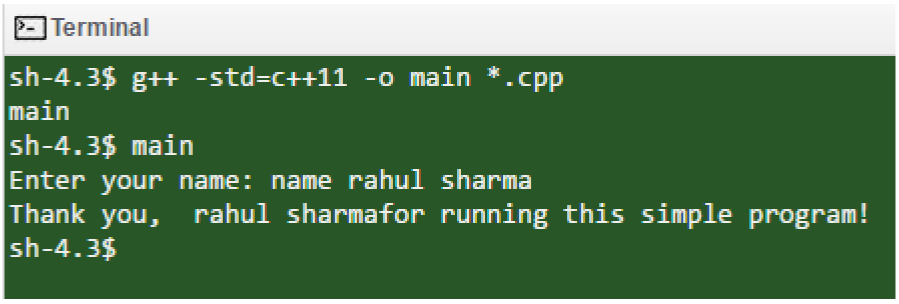
(i) Different Methods to Accept Strings

C++ strings allow you to directly initialize, assign, compare and reassign with the intuitive operators, as well as printing and reading (***For example,*** from the user), as shown in the ***example*** below:



```
#include<iostream>
using namespace std;
int main ()
{
string name;
cout<<"Enter your name: "<<flush;
cin>>name; // read string until the next separator
getline(cin, name); // read a whole line into the string name
if (name=="")
    {
        cout<<"You entered an empty string, "<<"assigning default\n";
        name="John";
    }
else
    {
        cout<<"Thank you, "<<name<<"for running this simple program!"<<endl;
    }
}
```

Output:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
main
sh-4.3$ main
Enter your name: name rahul sharma
Thank you,  rahul sharmafor running this simple program!
sh-4.3$
```

In the above program we used getline( ) function to read a whole line into the string "name". If the name is empty it takes the default value given in the if block, otherwise it assigns the user typed string. gets( ) and puts( ) functions are used to write and read a character. Now we will see the difference between cin and gets( ) as well as cout and puts().

Table 1.2.1: Difference between Gets and puts with cin and cout

Cin	gets( )
It can be used to take input of a value of any data type.	It can be used to take input of a string.
It takes the white space *i.e.*, a blank, a tab, or a new line character as a string terminator.	It does not take the white space *i.e.*, a blank, a tab, or a new line character, as a string terminator.
It requires header file iostream.h	It requires the header file stdio.h
For example, `char S[80];` `cout << "Enter a string:”;` `cin>>S;`	***For example,*** `char S[80];` `cout << "Enter a string:";` `gets(S);`

Table 1.2.2: String Handling Functions

Cout	puts( )
It can be used to display the value of any data type.	It can be used to display the value of a string.
It does not take a line feed after displaying the string.	It takes a line feed after displaying the string.
It requires the header file iostream.h	It requires the header file stdio.h
For example, `char S[80] = "Computers";` `cout<<S<<S;` **Output:** Computers Computers	***For example,*** `char S[80] = "Computers";` `puts(S);` `puts(S);` **Output:** Computers Computers

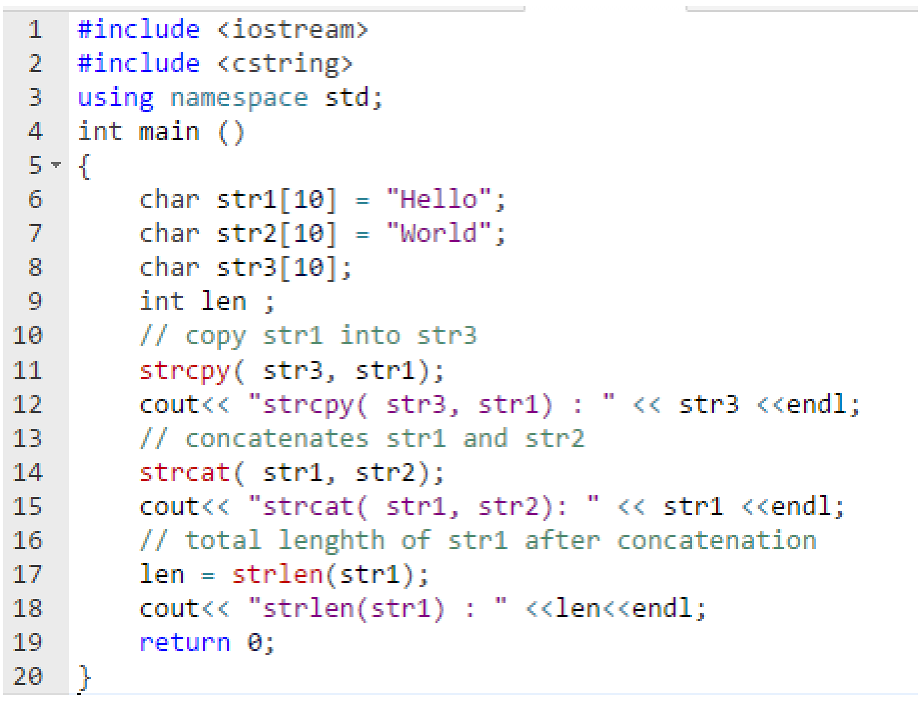
(ii) Different String Manipulations

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

Table 1.2.3: Various Functions and their Description

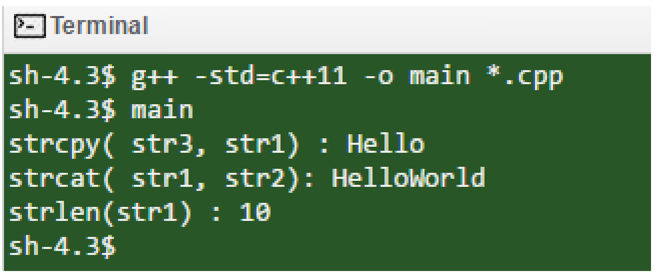
Function	Work of Function
strlen(s1)	Calculates the length of string
strcpy(s1,s2)	Copies a string to another string
strcat(s1,s2)	Concatenates(joins) two strings
strcmp(s1,s2)	Compares two string and Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

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



```
#include <iostream>
#include <cstring>
using namespace std;
int 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;
    return 0;
}
```

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

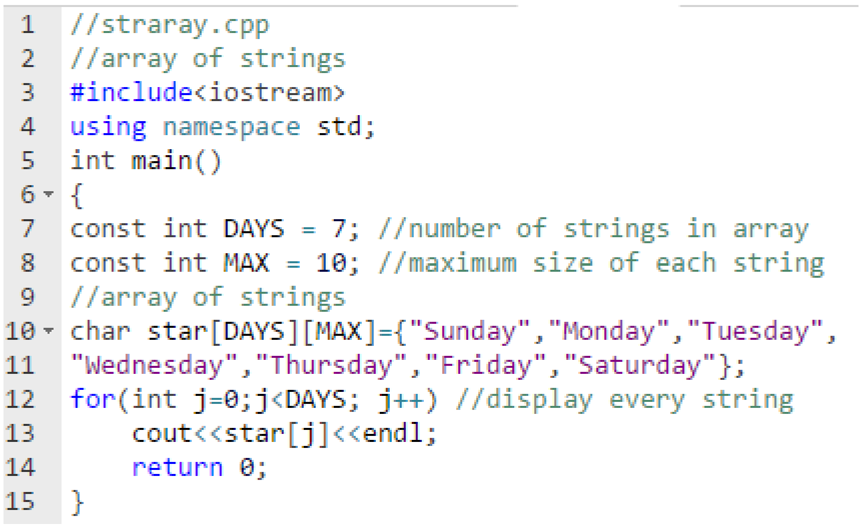


```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
sh-4.3$
```

(iii) Array of Strings

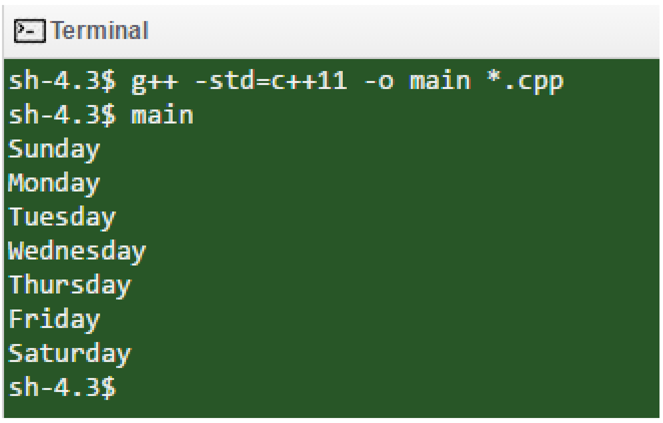
Array of strings in C++ is used to store a null terminated string which is a character array. This type of array has a string with a null character at the end of the string. Usually array of strings are declared one character long to accommodate the null character which marks the ending of any string. ***For example,*** if string is “ABC” then length will be 4 to store the null character also.

Here is an example to demonstrate the use of array of strings. The below program prints the names of days in a week stored in array of strings.



```cpp
//straray.cpp
//array of strings
#include<iostream>
using namespace std;
int main()
{
const int DAYS = 7; //number of strings in array
const int MAX = 10; //maximum size of each string
//array of strings
char star[DAYS][MAX]={"Sunday","Monday","Tuesday",
"Wednesday","Thursday","Friday","Saturday"};
for(int j=0;j<DAYS; j++) //display every string
    cout<<star[j]<<endl;
    return 0;
}
```

The program prints out each string from the array:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
sh-4.3$
```

In the above program, we have created 2 constant variables DAYS and MAX forming a 2D array of strings to store the days of the week. The array star is thus a 2 dimensional array which stores the names of seven days of the week. Variable DAYS specify the number of days in a week (*i.e.*, 7) and variable MAX specifies the maximum length of a string (assumes to be 9 for day named Wednesday). Using a 'for' loop all the day names are printed as output.

Self-assessment Questions

4) A string Hello world can be read using cin

   a) TRUE b) FALSE

5) Which function adds a string to the end of another string?

   a) Strat( ) b) stradd( )

   c) strcpy( ) d) strup( )

6) The gets( ) automatically appends a null character at the end of a string read from the keyboard

   a) TRUE b) FALSE

1.2.4 Structures and Unions

Array helps us to store data but it should be of same data type. It is often required to group data of different types and work with that grouped data as one single entity. We can achieve this grouping with a new data type called a structure.

Structures are a way of grouping many different values in variables of potentially different data types under the same name. This makes it a more modular and compact program.

A union is like a structure in which all the members' share the memory and these members can only be accessed one at a time. Unions prevent memory fragmentation by creating a standard size for certain data. Like structures, the members of unions can be accessed with the '.' and '->' operators.

(i) Basics of Structures

A structure is a convenient tool for handling a group of logically related data items.

For example, roll number, marks and percentage are all related data items under a common group storing student's results. Structure helps to organize complex data in a more meaningful way.

A structure is a combination of variables of different data types under a single name. Let us take the example of a book. It may be required to store details of 5 books. So a structure "book" could be created that would hold all of the information about a single book including name, title, authors, publishers etc. All these variables will be declared under a single name "book". Let us see how to declare a structure book.

(ii) Declaring and Defining a Structure

We can declare a structure using struct as a key word. The above example can be declared as below.

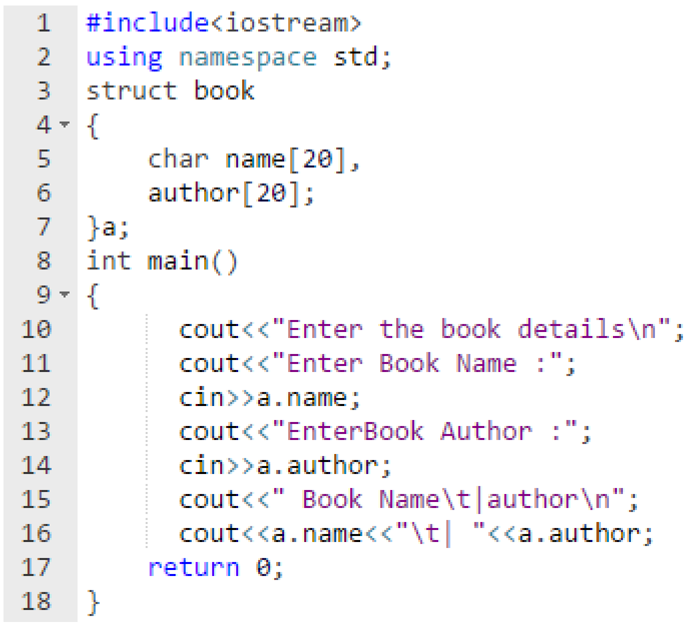
```
struct Book
{
      char Name[100];
      char Author[100];
      char Publisher[80];
      int Year;
};
```

The keyword struct defines a book and each line with in the braces defines the elements of the Book. Now whenever we create an instance of Book it will have all the elements of the structure *i.e.,* Name, Author, Publisher and Year.

We can declare the structure

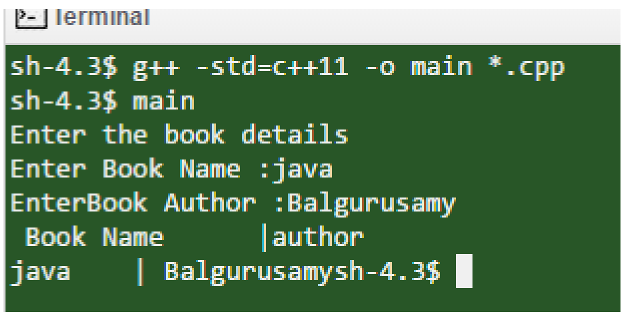
```
Book CProgrammingBook;
Let us initialize the structure variable CProgrammingBook.
Book CProgrammingBook =
{
      "Beginning VC++ 6",
      "Ivor Horton",
      "Wrox",
      2001
};
```

For example,



```
#include<iostream>
using namespace std;
struct book
{
    char name[20],
    author[20];
}a;
int main()
{
        cout<<"Enter the book details\n";
        cout<<"Enter Book Name :";
        cin>>a.name;
        cout<<"EnterBook Author :";
        cin>>a.author;
        cout<<" Book Name\t|author\n";
        cout<<a.name<<"\t| "<<a.author;
    return 0;
}
```

Output:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Enter the book details
Enter Book Name :java
EnterBook Author :Balgurusamy
 Book Name       |author
java     | Balgurusamysh-4.3$
```

(iii) Accessing Structure Members

We can access all the members of a structure by adding a period after the name of the structure variable name and then the name of the field we want to access. ***For example,*** we want to change the Publishing year from 2001 to 2002, we will do it as

```
CProgrammingBook.Year = 2002;
```

(iv) Array of Structures

We can declare arrays of structure as well as array of structure pointers.

As you know, structure is a collection of different data types (variables) which are grouped together. An array of structures is nothing but collection of structures. An array of structures refers to an array in which each element is of structure type. To declare an array of structures, firstly, a structure is defined and then an array of that structure is declared. Arrays of structures and arrays of structure pointers are also possible. Two groups of formats exist. Here is the first one.

```
   structstruct_name name[size] = { init_list };
   structstruct_name name[size1][size2][sizeN] = { init_list };
   structstruct_name *pname[size] = { init_list };
   structstruct_name *pname[size1][size2][sizeN] = { init_list };
```

The keyword struct is optional when you define struct_name elsewhere and the rules for size are the same as the rules for arrays of built-in types (see "Arrays" on page 35). The optional brace enclosing init_list initializes a data member in each array element. With arrays of structure pointers (pname), init_list must have pointer expressions that match or convert to struct_name. When you initialize arrays of structures or arrays of structure pointers, you must include braces with init_list.

The second group of formats define a structure type and follow the same rules as above. The keyword struct must appear in both formats, but struct_name is optional.

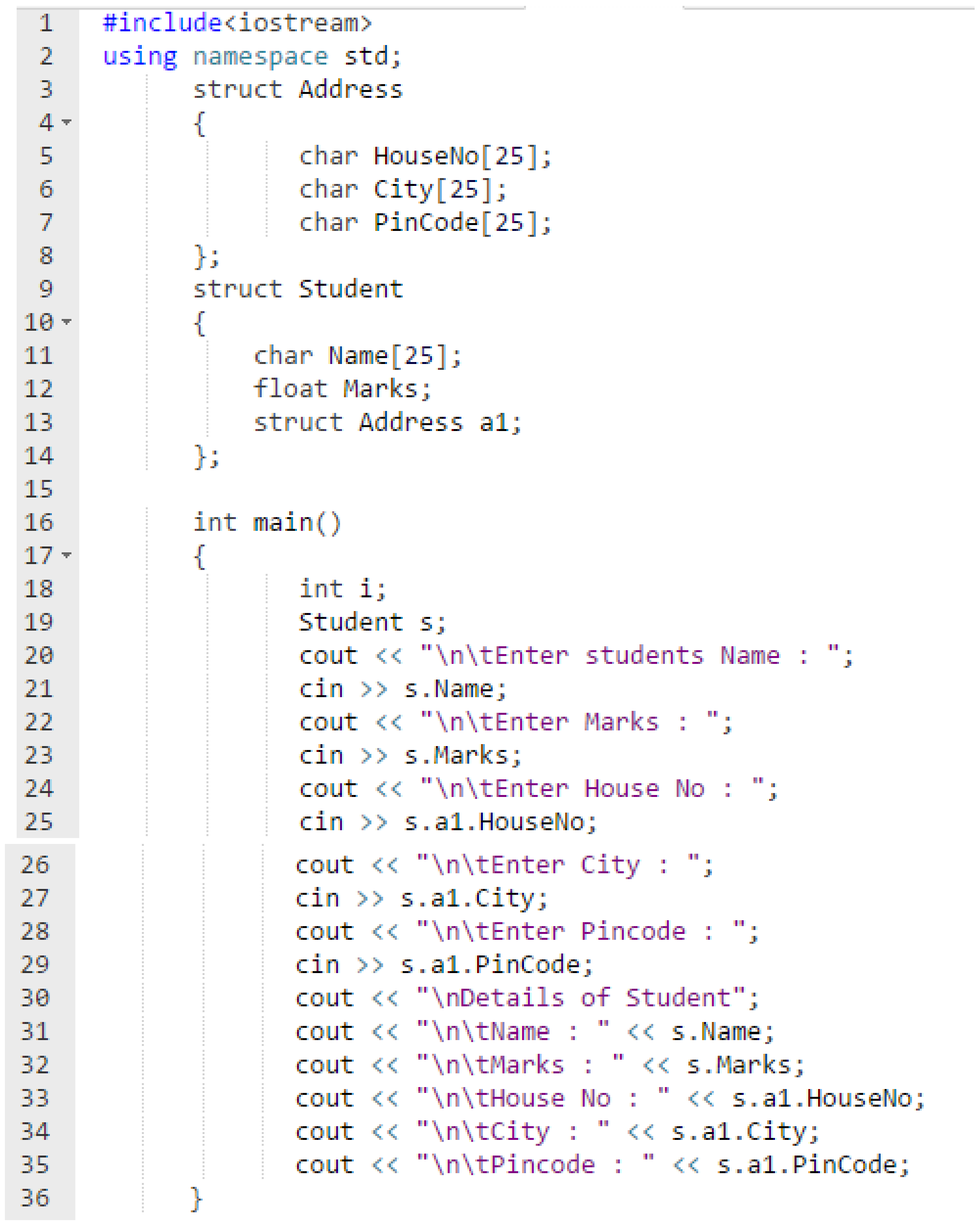
```
Struct struct_name
{
   Type data_member;
   Type member_function(signature);
} name[size1][size2][sizeN] = { init_list };
structstruct_name
{
   Type data_member;
   Type member_function(signature);
} *pname[size1][size2][sizeN] = { init_list };
```

Like other user defined data types, a structure can also include another structure, as its member, in its definition. A structure defined within another structure is known as a **nested structure**.

Syntax for structure within structure or nested structure:

```
struct structure1
{
      -------------
      -------------
};
struct structure2
{
      -------------
      -------------
      struct structure1 obj;
};
```

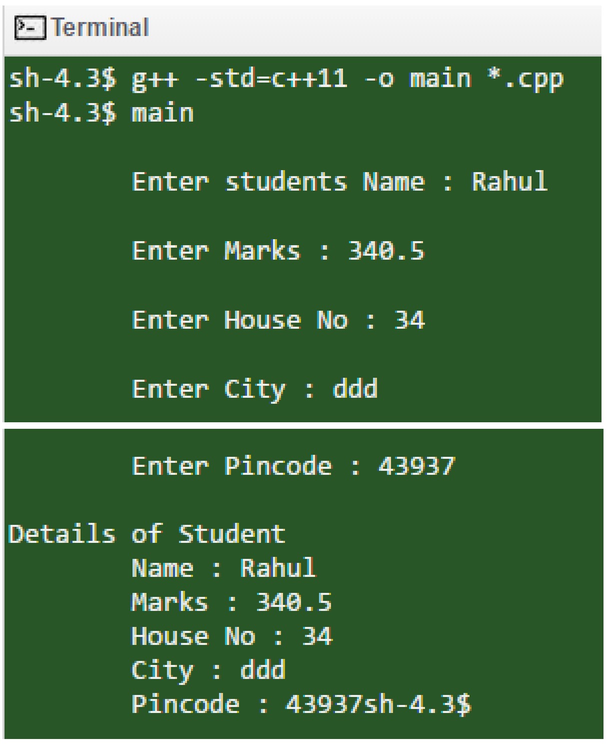
Let us discuss the concept of nested structure with an example of a code segment.



```
#include<iostream>
using namespace std;
    struct Address
    {
            char HouseNo[25];
            char City[25];
            char PinCode[25];
    };
    struct Student
    {
        char Name[25];
        float Marks;
        struct Address a1;
    };

    int main()
    {
            int i;
            Student s;
            cout << "\n\tEnter students Name : ";
            cin >> s.Name;
            cout << "\n\tEnter Marks : ";
            cin >> s.Marks;
            cout << "\n\tEnter House No : ";
            cin >> s.a1.HouseNo;
            cout << "\n\tEnter City : ";
            cin >> s.a1.City;
            cout << "\n\tEnter Pincode : ";
            cin >> s.a1.PinCode;
            cout << "\nDetails of Student";
            cout << "\n\tName : " << s.Name;
            cout << "\n\tMarks : " << s.Marks;
            cout << "\n\tHouse No : " << s.a1.HouseNo;
            cout << "\n\tCity : " << s.a1.City;
            cout << "\n\tPincode : " << s.a1.PinCode;
    }
```

Output:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main

        Enter students Name : Rahul

        Enter Marks : 340.5

        Enter House No : 34

        Enter City : ddd

        Enter Pincode : 43937

Details of Student
        Name : Rahul
        Marks : 340.5
        House No : 34
        City : ddd
        Pincode : 43937sh-4.3$
```

In the above example two structures are created. Structure Student stores the students' details whereas structure Address stores the address details of student. The structure Student contains a structure variable of data type structure; Address as one of its member using the below given line.

```
struct Address a1;
```

The structure Student contains the structure Address as address additionally has house no, city and pincode. Hence structure Student is external structure and Address is internal structure.

Thus a single variable inside the structure Address can be initialized as:

```
s.a1.houseno=156;
```

You can also initialize nested structure members together using either of these 2 lines.

```
student s={“rahul”, 340, 45, “Panjim”, 403838};
                     or
student s={“rahul”, 340, {45, “Panjim”, 403838}};
```

The members of nested structure can be accessed using this statement.

```
Cout<<s.a1.houseno;  //Display house number of student s
```

(v) Unions

A union is a user defined variable which holds members of different sizes and type designed to save memory. Union uses a single memory location to hold more than one variable. A union is a user-defined data type in which all members share the same memory location. It means that no matter how many members a union has, it always uses only enough memory to store the largest member. Unions can be useful for saving memory when you have lots of objects and limited memory. We can use unions in following situations:

- Share a single memory location for a variable myVar1 and use the same location for myVar2 of different data type when myVar1 is not required any more.
- Use it if you want to user, ***for example,*** a long variable as two short type variables.
- We don't know what type of data is to be passed to a function and you pass union which contains all the possible data types included in it.

Defining a Union

Union can be defined by the keyword union as shown below.

```
Union Sample
{
   int a;
   float b;
};
```

Here we have defined a union with the name Sample and it has two members *i.e.,* a of type int and b of type long

Declaring the Union

We can declare the union in various ways.

First Method:

```
Union Sample
{
   int a;
   long b;
}s1;
```

So s1 will be the variable of type Sample.

Second Method:

```
Sample s1;
```

Initializing the Union

We can initialize the union in various ways. ***For example,***

```
Union Sample
{
    int a;
    long b;
}s1={3,4};
```

Or we can initialize it as

```
s1.a=3;
s1.b=4
```

In later stages we can also initialize the b, but this will overwrite a value. Normally when we declare the union it is allocated with the memory that can be occupied by its biggest member. In this example s1 will occupy the memory which a long type variable can occupy.

(vi) Difference between Structures and Unions

The following table 1.2.4 shows the differences between structures and unions.

Table 1.2.4: Difference between Structures and Unions

Structures	Unions
A structure is defined using a “struct” keyword.	A union is defined using a “union” keyword.
Structure occupies appropriate separate memory for all of its members	Union occupies memory for that member which needs largest chunk of bytes to be stored.
A structure enables us treat a number of different variables stored at different addresses in memory.	A union enables us to treat the same space in memory as a number of different variables.
Several members of a structure can be initialized at once.	Only the first member of a union can be initialized at one time.
The address of each member will be in ascending order which indicates that memory for each member will start at different offset values.	The address is same for all the members of a union which indicates that every member begins at the same offset value.

(vii) Enumerated Data Types

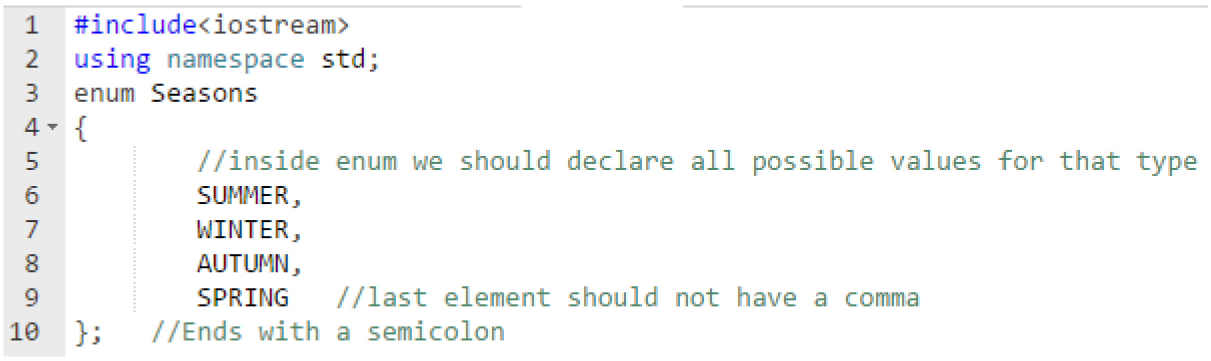
C++ allows programmers to create their own data types. You can use enumerated data type to do this. An **enumerated type** (also called an **enumeration**) is a data type where every possible value is defined as a symbolic constant (called an **enumerator**).

For example, an enum named "seasons" can have only one of the 4 values spring, summer, autumn and winter.

- **Declaration**

Enumerations are declared via the enum keyword. Let us look into an example:

// Declare a new enumeration named Seasons



```
#include<iostream>
using namespace std;
enum Seasons
{
        //inside enum we should declare all possible values for that type
        SUMMER,
        WINTER,
        AUTUMN,
        SPRING   //last element should not have a comma
};   //Ends with a semicolon
```

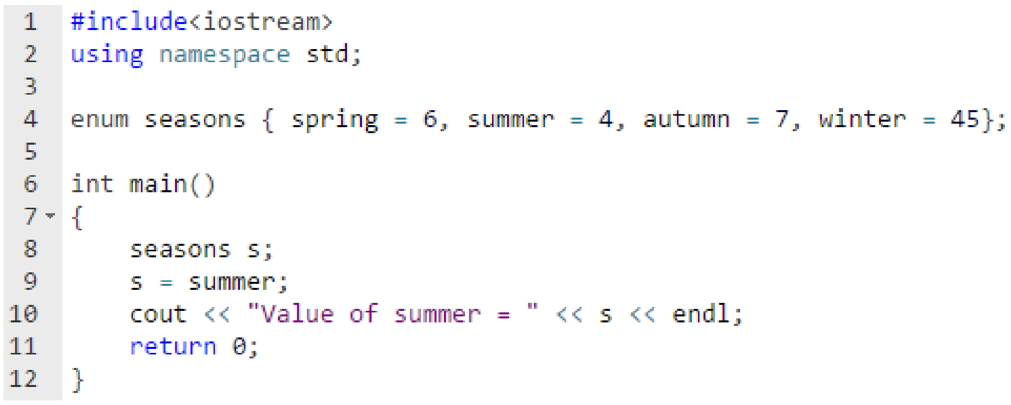
We can define variables of enumerated type seasons as given below:

Seasons value=WINTER;

Declaring an enumeration does not allocate any memory. When a variable of the numerated type is defined (such as variable value in the example above), memory is allocated for that variable at that time.

Each enumerator is separated by a comma and the entire enumeration is ended with a semicolon.

Example:

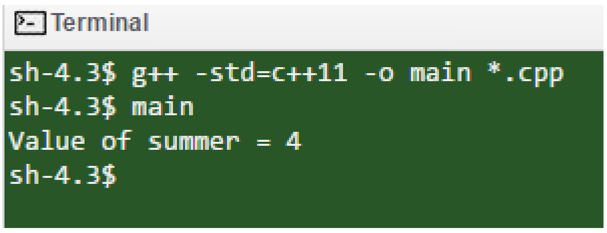


```
#include<iostream>
using namespace std;

enum seasons { spring = 6, summer = 4, autumn = 7, winter = 45};

int main()
{
    seasons s;
    s = summer;
    cout << "Value of summer = " << s << endl;
    return 0;
}
```

Output:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Value of summer = 4
sh-4.3$
```

Naming enums

Enum identifiers are often named starting with a capital letter and the enumerators should be written in capital letters. Because enumerators are placed into the same namespace as the enumeration, an enumerator name can't be used in multiple enumerations within the same namespace:

```
enum Color
{
   RED,
   BLUE, // BLUE is put into the global namespace
   GREEN
};

enum Feeling
{
   HAPPY,
   TIRED,
   BLUE // error, BLUE was already used in enum Color in the global
   namespace
};
```

Consequently, it's common to prefix enumerators with a standard prefix like ANIMAL_ or COLOR_, both to prevent naming conflicts and for code documentation purposes.

- ## Usage of Enumerated data types

  - Enumerated types are incredibly useful for code documentation and readability purposes when you need to represent a specific, predefined set of states.

  - ***For example***, functions often return integers to the caller to represent error codes when something went wrong inside the function. Typically, small negative numbers are used to represent different possible error codes.

    For example,

```
intreadFileContents()
{
    if (!openFile())
        return -1;
    if (!readFile())
        return -2;
    if (!parseFile())
        return -3;

    return 0; // success
}
```

  - However, using magic numbers like this isn't very descriptive. An alternative method would be through use of an enumerated type:

```
enumParseResult
{
    SUCCESS = 0,
    ERROR_OPENING_FILE = -1,
    ERROR_READING_FILE = -2,
    ERROR_PARSING_FILE = -3
};

ParseResultreadFileContents()
{
    if (!openFile())
        return ERROR_OPENING_FILE;
    if (!readFile())
        return ERROR_READING_FILE;
    if (!parsefile())
        return ERROR_PARSING_FILE;

    return SUCCESS;
}
```

- This is much easier to read and understand than using magic number return values. Furthermore, the caller can test the function's return value against the appropriate enumerator, which is easier to understand than testing the return result for a specific integer value.

```
if (readFileContents() == SUCCESS)
    {
       // do something
    }
else
    {
       // print error message
    }
```

- Enumerated types are best used when defining a set of related identifiers. ***For example,*** Let us say you were writing a game where the player can carry one item, but that item can be several different types. You could do this:

```
#include <iostream>
#include <string>

enumItemType
{
    ITEMTYPE_SWORD,
    ITEMTYPE_TORCH,
    ITEMTYPE_POTION
};

std::string getItemName(ItemTypeitemType)
{
    if (itemType == ITEMTYPE_SWORD)
        return std::string("Sword");
    if (itemType == ITEMTYPE_TORCH)
        return std::string("Torch");
    if (itemType == ITEMTYPE_POTION)
        return std::string("Potion");
}

int main()
{
    // ItemType is the enumerated type we've declared above.
     // itemType (lower case i) is the name of the variable we're
defining (of type ItemType).
     // ITEMTYPE_TORCH is the enumerated value we're initializing
variable itemType with.
    ItemTypeitemType(ITEMTYPE_TORCH);
```

```
     std::cout<< "You are carrying a " <<getItemName(itemType) <<
"\n";

    return 0;
}
```

Or alternatively, if you were writing a function to sort a bunch of values:

```
enumSortType
{
    SORTTYPE_FORWARD,
    SORTTYPE_BACKWARDS
};

void sortData(SortType type)
{
    if (type == SORTTYPE_FORWARD)
        // sort data in forward order
    else if (type == SORTTYPE_BACKWARDS)
        // sort data in backwards order
}
```

Many languages use Enumerations to define booleans. A boolean is essentially just an numeration with 2 enumerators: false and true! However, in C++, true and false are defined as keywords instead of enumerators.

Did you Know?

Class can also be declared inside namespace and defined outside namespace

Self-assessment Questions

7) A structure can be placed within another structure and is known as

a) Parallel structure	b) Self-referential structure
c) Nested structure	d) two structure

8) A union can have another union as its member

a) TRUE	b) FALSE

9) The identifiers in enumerated type are automatically assigned values

a) TRUE	b) FALSE

Summary

- An array is used to store a collection of data items which are all of the same type. An individual element of an array is accessed by its index, which must be an integer. The first element in the array has index 0.

- When arrays are declared they must be given an integer number of elements. This number of elements must be known to the compiler at the time the array is declared.

- Arrays can be initialised when they are declared. When an array is passed as a parameter to a function then it is passed as a reference parameter. Hence any changes made to the array inside the function will change the actual array that is passed as a parameter.

- Character strings are represented by arrays of characters. The string is terminated by the null character. In declaring a string you must set the size of the array to at least one longer than required to hold the characters of the string to allow for this null character. Strings can be input and output via the input and output streams, cin and cout.

- An enumeration is a programmer-defined type that is limited to a fixed list of values. A declaration gives the type a name and specifies the permissible values, which are called enumerators. Definitions can then create variables of this type. Internally the compiler treats enumeration variables as integers.

- An array is used to store a collection of data items which are all of the same type. An individual element of an array is accessed by its index, which must be an integer. The first element in the array has index 0.

- When arrays are declared they must be given an integer number of elements. This number of elements must be known to the compiler at the time the array is declared.

- Arrays can be initialised when they are declared. When an array is passed as a parameter to a function then it is passed as a reference parameter. Hence any changes made to the array inside the function will change the actual array that is passed as a parameter.

- Character strings are represented by arrays of characters. The string is terminated by the null character. In declaring a string you must set the size of the array to at

least one longer than required to hold the characters of the string to allow for this null character. Strings can be input and output via the input and output streams, cin and cout.

- An enumeration is a programmer-defined type that is limited to a fixed list of values. A declaration gives the type a name and specifies the permissible values, which are called enumerators. Definitions can then create variables of this type. Internally the compiler treats enumeration variables as integers.

Terminal Questions

1. Explain the fundamentals of array.
2. Explain different string manipulation functions.
3. Differentiate between structure and union.
4. Explain array of structures.
5. Illustrate the declaration and definition of enumerated data types.

Answer Keys

Self-assessment Questions	
Question No.	**Answer**
1	a
2	a
3	c
4	b
5	a
6	a
7	c
8	a
9	a

Activity

Activity Type: Online **Duration:** 45 Minutes

Description:

1. Develop and execute a program on array of structures to read 5 students data like

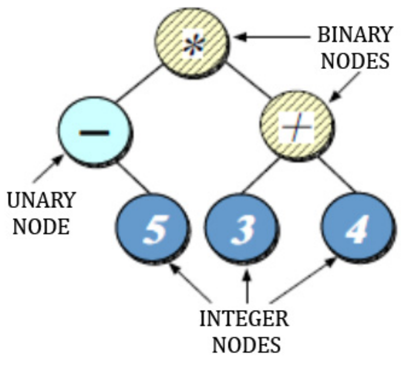
   - USN
   - Name
   - DOB
   - Marks
   - Mobile number

   and display the same on the output screen

2. Develop and execute a program to read two matrices A (M x N) and B (P x Q) and to compute the product of A and B if the matrices are compatible for multiplication. The program is to print the input matrices and the resultant matrix with suitable headings and format if the matrices are compatible for multiplication, otherwise the program must print a suitable message. (For the purpose of demonstration, the array sizes M, N, P and Q can all be less than or equal to 3)

Case Study

Expression trees consist of nodes containing operators and operands. Operators have different precedence levels, different associativity's and different arities. ***For example,*** Multiplication takes precedence over addition. The multiplication operator has two arguments, whereas unary minus operator has only one. Operands are integers, doubles, variables, etc.



Expression trees

Trees may be "evaluated" via different traversals ***for example,*** in-order, post-order, pre-order, level-order. The evaluation step may perform various operations, ***For example,*** Traverse and print the expression tree, return the "value" of the expression tree, Generate code, Perform semantic analysis.

```
// A C++ program to demonstrate use of class
// in a namespace
#include <iostream>
usingnamespacestd;

namespacens
{
    // Only declaring class here
    classgeek;
}

// Defining class outside
classns::geek
{
public:
    voiddisplay()
    {
        cout<< "ns::geek::display()\n";
    }
};

intmain()
{
    //Creating Object of student Class
    ns::geekobj;
    obj.display();
    return0;
}
```

Output:

```
   ns::geek::display()
```

Bibliography

e-References

- Ocw.mit.edu,(2016). *Introduction to C++: Arrays and strings.* Retrieved 7 April, 2016 from http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/lecture-notes/MIT6_096IAP11_lec04.pdf

- pages.cpsc.ucalgary.ca ,(2016). *Index of /~jacob/Courses/Fall00/CPSC231/Slides.* Retrieved 7 April, 2016 from http://pages.cpsc.ucalgary.ca/~jacob/Courses/Fall00/CPSC231/Slides/10-ArraysAndStrings.pdf

- cplusplus.com,(2016). *Character sequences*. Retrieved 7 April, 2016 from www.cplusplus.com/doc/tutorial/ntcs/

- mycplus.com,(2016). *Unions and Structures.* Retrieved 7 April, 2016 from http://www.mycplus.com/tutorials/c-programming-tutorials/unions-structures/

External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer* (3rd ed.). Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++* (3rd ed.). Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer* (1st ed.). Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language* (3rd ed.). Pearson Publications.

Video Links

Topic	Link
Array Fundamentals	https://www.youtube.com/watch?v=ZLk7qV9wEDw
Strings	https://www.youtube.com/watch?v=_RYLJoJBC7k
Structures and unions	https://www.youtube.com/watch?v=2ooUkhIuKew
Enumerated Datatypes	https://www.youtube.com/watch?v=pkQ7S_zOSbE
Array Fundamentals	https://www.youtube.com/watch?v=ZLk7qV9wEDw

Notes: