About the content

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming

UNIT-II
Module-1

C++ References

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are −

- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Creating References in C++

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example −

```
int i = 17;
```

We can declare reference variables for i as follows.

```
int& r = i;
```

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double −

```
#include <iostream>
```

```
using namespace std;

int main () {
  // declare simple variables
  int    i;
  double d;

  // declare reference variables
  int&    r = i;
  double& s = d;

  i = 5;
  cout << "Value of i : " << i << endl;
  cout << "Value of i reference : " << r  << endl;

  d = 11.7;
  cout << "Value of d : " << d << endl;
  cout << "Value of d reference : " << s  << endl;

  return 0;
}
```

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

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer −

C++ Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double; // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

```
double* pvalue = NULL;
if( !(pvalue )) {
cout << "Error: out of memory." <<endl;
exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows −

```
delete   pvalue; // Release memory pointed to by pvalue
```

Initialize memory: We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

pointer-variable = **new** data-type(value);

Example:

*int *p = new int(25);*
*float *q = new float(75.25);*

Let us put above concepts and form the following example to show how 'new' and 'delete' work −

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

int main () {
  double* pvalue  = NULL; // Pointer initialized with null
  pvalue  = new double;   // Request memory for the variable

  int *q = new int (25); // Pointer initialized with value

  *pvalue = 29494.99;     // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;
  cout << "Value of q : " << *q << endl;


  delete pvalue;        // free up the memory.
  delete q;
```

```
   return 0;
}
```

If we compile and run above code, this would produce the following result −

Value of pvalue : 29495

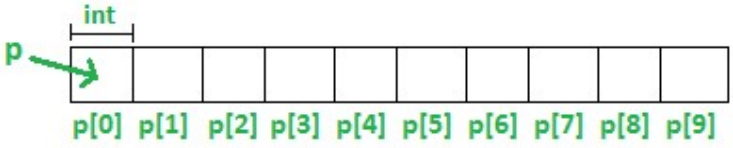
Dynamic Memory Allocation for Arrays

Allocate a block of memory: new operator is also used to allocate a block(an array) of memory of type *data-type*. .

Example:

int *p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.

To free the dynamically allocated array pointed by pointer-variable, use the following form of *delete*:

delete[] pointer-variable; // Release block of memory pointed by pointer-variable

Example:

delete[] p; // It will free the entire array pointed by p.

```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{

   // Request block of memory of size n
   int n = 5;
   int *q = new  int[n];

   if (!q)
      cout << "allocation of memory failed\n";
   else
```

```
    {
       for (int  i = 0; i < n; i++)
          q[i] = i+1;  // or *(q+i)=i+1

       cout << "Values stored in block of memory: ";
       for (int  i = 0; i < n; i++)
          cout << q[i] << " ";
    }

    // freed the block of allocated memory
    delete [] q;

    return 0;
}
```

Output:

```
Value store in block of memory: 1 2 3 4 5
```

C++ structures

- The structure is a user-defined data type that is available in C++.
- Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.
- A structure is declared by using the keyword “struct“. When we declare a variable of the structure we need to write the keyword “struct in C language but for C++ the keyword is not mandatory

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

```
struct [structure tag] {
member definition;
member definition;
...
member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
    string title;
    string author;
    float price;
```

```
    int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure −

```
#include <iostream>
#include <string>
using namespace std;
void printBook( struct Books book );
struct Books {
  string  title;
  string  author;
  float  price;
  int   book_id;
  };

int main() {
  struct Books Book1;       // Declare Book1 of type Book
  struct Books Book2;       // Declare Book2 of type Book

  // book 1 specification
  Book1.title="Learn C++ Programming";
  Book1.author="Chand Miyan";
  Book1.price= 954.55;
  Book1.book_id = 6495407;

  // book 2 specification
  Book2.title="Telecom Billing";
  Book2.author="Yakit Singha";
  Book2.price= 555.9999;
  Book2.book_id = 6495700;

  // Print Book1 info
  cout << "Book 1 title : " << Book1.title <<endl;
  cout << "Book 1 author : " << Book1.author <<endl;
  cout << "Book 1 price : " << Book1. price <<endl;
  cout << "Book 1 id : " << Book1.book_id <<endl;

  // Print Book2 info
  cout << "Book 2 title : " << Book2.title <<endl;
  cout << "Book 2 author : " << Book2.author <<endl;
  cout << "Book 2 price: " << Book2. price <<endl;
  cout << "Book 2 id : " << Book2.book_id <<endl;

  return 0;
}
```

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

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 price : 954.55
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 price : 555.9999
Book 2 id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example −

```
#include <iostream>
#include <string>
using namespace std;
void printBook( struct Books book );
struct Books {
   string  title;
   string  autho;
   string  subject;
   int   book_id;};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   Book1.title="Learn C++ Programming";
   Book1.author="Chand Miyan";
   Book1.subject="C++ Programming";
   Book1.book_id = 6495407;

   // book 2 specification
   Book2.title="Telecom Billing";
   Book2.author="Yakit Singha";
   Book2.subject= "Telecom";
   Book2.book_id = 6495700;

   // Print Book1 info
   printBook( Book1 );
   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book ) {
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
```

```
   cout << "Book id : " << book.book_id <<endl;
}
```

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

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

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

using namespace std;
void printBook( struct Books *book );

struct Books {
   string  title;
   string  autho;
   string  subject;
   int   book_id;
};
int main() {
   struct Books  Book1;        // Declare Book1 of type Book
   struct Books  Book2;        // Declare Book2 of type Book

   // Book 1 specification
   Book1.title= "Learn C++ Programming";
   Book1.author= "Chand Miyan";
   Book1.subject= "C++ Programming";
   Book1.book_id = 6495407;

   // Book 2 specification
   Book2.title="Telecom Billing";
```

```
   Book2.author="Yakit Singha";
   Book2.subject="Telecom";
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

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

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Difference Between C Structures and C++ Structures

C Structures	C++ Structures
Only data members are allowed, it cannot have member functions.	Can hold both: member functions and data members.
Cannot have static members.	Can have static members.
Cannot have a constructor inside a structure.	Constructor creation is allowed.
Direct Initialization of data members is not possible.	Direct Initialization of data members is possible.
Writing the ‘struct’ keyword is necessary to declare structure-type variables.	Writing the ‘struct’ keyword is not necessary to declare structure-type variables.
Do not have access modifiers.	Supports access modifiers.

C Structures	C++ Structures
Only pointers to structs are allowed.	Can have both pointers and references to the struct.
Sizeof operator will generate 0 for an empty structure.	Sizeof operator will generate 1 for an empty structure.
Data Hiding is not possible.	Data Hiding is possible.

Similarities Between the C and C++ Structures

- Both in C and C++, members of the structure have public visibility by default.

Lets discuss some of the above mentioned differences and similarities one by one:

1. Member functions inside the structure: Structures in C cannot have member functions inside a structure but Structures in C++ can have member functions along with data members.

```
// C Program to Implement Member functions inside structure

#include <stdio.h>

struct marks {

    int num;

    // Member function inside Structure to
    // take input and store it in "num"
    void Set(int temp) { num = temp; }

    // function used to display the values
    void display() { printf("%d", num); }
};

// Driver Program
int main()
{
    struct marks m1;
    // calling function inside Struct to
    // initialize value to num
    m1.Set(9);

    // calling function inside struct to
    // display value of Num
    m1.display();
}
```

Output

```
This will generate an error in C but no error in C++.
```

```
// C++ Program to Implement Member functions inside
// structure

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

```
struct marks {
    int num;

    // Member function inside Structure to
    // take input and store it in "num"
    void Set(int temp) { num = temp; }

    // function used to display the values
    void display() { cout << "num=" << num; }
};

// Driver Program
int main()
{
    marks m1;

    // calling function inside Struct to
    // initialize value to num
    m1.Set(9);

    // calling function inside struct to
    // display value of Num
    m1.display();
}
```

Output

num=9

2. Static Members: C structures cannot have static members but are allowed in C++.

```
// C program with structure static member

struct Record {
    static int x;
};

// Driver program
int main() { return 0; }
```

This will generate an error in C but not in C++.

3. Constructor creation in structure: Structures in C cannot have a constructor inside a structure but Structures in C++ can have Constructor creation.

```
// C program to demonstrate that
// Constructor is not allowed

#include <stdio.h>

struct Student {
    int roll;
    Student(int x) { roll = x; }
};

// Driver Program
int main()
```

```
{
    struct Student s(2);
    printf("%d", s.x);
    return 0;
}
```

This will generate an error in C.
Output in C++:
2

4. Using struct keyword: In C, we need to use a struct to declare a struct variable. In C++, a struct is not necessary. For example, let there be a structure for Record. In C, we must use "struct Record" for Record variables. In C++, we need not use struct, and using 'Record' only would work.

5. Access Modifiers: C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language
.
6. Pointers and References: In C++, there can be both pointers and references to a struct in C++, but only pointers to structs are allowed in C.

7. sizeof operator: This operator will generate **0** for an empty structure in C whereas **1** for an empty structure in C++.

```
// C program to illustrate empty structure

#include <stdio.h>

// empty structure
struct Record {
};

// Driver Code
int main()
{
    struct Record s;
    printf("%lu\n", sizeof(s));
    return 0;
}
```

Output in C:
0

Output in C++:
1

NOTE: *The default type of sizeof is long unsigned int , that's why "%lu" is used instead of "%d" in printf function.*
8. Data Hiding: C structures do not allow the concept of Data hiding but are permitted in C++ as it is an object-oriented language whereas C is not.

Unions

A union is a type of structure that can be used where the amount of memory used is a key factor.

- Similarly to the structure, the union can contain different types of data types.
- Each time a new variable is initialized from the union it overwrites the previous in C language but in C++ we also don't need this keyword and uses that memory location.
- This is most useful when the type of data being passed through functions is unknown, using a union which contains all possible data types can remedy this problem.
- It is declared by using the keyword "**union**".

Below is the C++ program illustrating the implementation of union:

```
// C++ program to illustrate the use
// of the unions
#include <iostream>
using namespace std;

// Defining a Union
union GFG {
    int Geek1;
    char Geek2;
    float Geek3;
};

// Driver Code
int main()
{
    // Initializing Union
    union GFG G1;

    G1.Geek1 = 34;

    // Printing values
    cout << "The first value at "
         << "the allocated memory : " << G1.Geek1 << endl;

    G1.Geek2 = 34;

    cout << "The next value stored "
         << "after removing the "
         << "previous value : " << G1.Geek2 << endl;

    G1.Geek3 = 34.34;

    cout << "The Final value value "
         << "at the same allocated "
         << "memory space : " << G1.Geek3 << endl;
    return 0;
}
```

Output

The first value at the allocated memory : 34

The next value stored after removing the previous value : "

The Final value value at the same allocated memory space : 34.34

Explanation: In the above code, Geek2 variable is assigned an integer (34). But by being of char type, the value is transformed through coercion into its char equivalent (“). This result is correctly displayed in the Output section.

Module-2
Classes and Objects

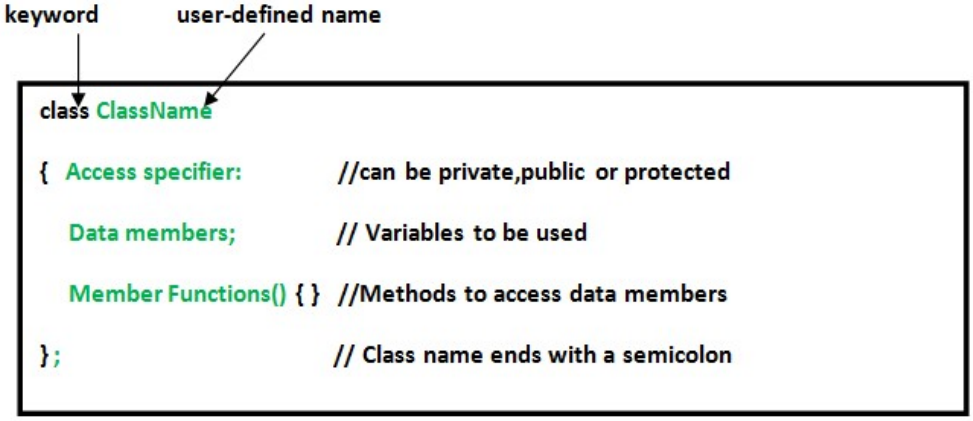
Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Access Data Members and Member Functions

We can access the data members and member functions of a class by using a . (dot) operator.

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Example 1: Object and Class in C++ Programming

```
// Program to illustrate the working of
// objects and class in C++ Programming

#include <iostream>
using namespace std;

// create a class
class Room {

  public:
   double length;
   double breadth;
   double height;

   double calculateArea() {
      return length * breadth;
   }

   double calculateVolume() {
      return length * breadth * height;
   }
};

int main() {

   // create object of Room class
   Room  room1;

   // assign values to data members
   room1.length = 42.5;
   room1.breadth = 30.8;
   room1.height = 19.2;

   // calculate and display the area and volume of the room
```

```
    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

In this program, we have used the `Room` class and its object `room1` to calculate the area and volume of a room.
In `main()`, we assigned the values of `length`, `breadth`, and `height` with the code:

```
room1.length = 42.5;
room1.breadth = 30.8;
room1.height = 19.2;
```

We then called the functions `calculateArea()` and `calculateVolume()` to perform the necessary calculations.
Note the use of the keyword `public` in the program. This means the members are public and can be accessed anywhere from the program.

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

```
class Base {
public:
// public members go here
protected:

// protected members go here
private:
// private members go here
};
```

The public Members

A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the previous program

The private Members

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

By default all the members of a class would be private, for example in the following class y is a private member, which means until you label a member, it will be assumed a private member −

```
class MyClass {
   int a;   // Private attribute
  public:    // Public access specifier
   int x;   // Public attribute
  private:   // Private access specifier
   int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.a = 10;  // Not allowed (private)
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

If you try to access a private member, an error occurs:

error: y is private

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

Example 2: Using public and private in Class

```
// Program to illustrate the working of
// public and private in C++ Class
#include <iostream>
using namespace std;

class Room {

  private:
   double length;
   double breadth;
   double height;

  public:
   // function to initialize private variables
   void initData(double len, double brth, double hgt) {
      length = len;
      breadth = brth;
```

```
        height = hgt;
    }
    double calculateArea() {
        return length * breadth;
    }
    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;
    // pass the values of private variables as arguments
    room1.initData(42.5, 30.8, 19.2);

    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

The above example is nearly identical to the first example, except that the class variables are now private.

Since the variables are now private, we cannot access them directly from `main()`. Hence, using the following code would be invalid:

```
// invalid code
obj.length = 42.5;
obj.breadth = 30.8;
obj.height = 19.2;
```

Instead, we use the public function `initData()` to initialize the private variables via the function parameters `double len`, `double brth`, and `double hgt`.

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Member functions can be defined within the class definition or separately using **scope resolution operator, :** −. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier.

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows −

Let us put above concepts to Store and Display Employee Information

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member
     string name; //data member
     float salary; //data member
//function definition inside class(inline function)
     void insert(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }

void display()    //function declared inside class

};

//function definition outside class
void Employee::display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }

int  main(void) {
   Employee e1; //creating an object of Employee
   Employee e2; //creating an object of Employee
   e1.insert(201, "Sonoo", 990000);
   e2.insert(202, "Nakul", 29000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.
Note: Declaring a friend function is a way to give private access to a non-member function.

Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

There can be two types of constructors in C++.
1.Default constructor
2.Parameterized constructor

C++ Default Constructor
A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
  public:
      Employee()
      {
         cout<<"Default Constructor Invoked"<<endl;
      }
};
int main(void)
{
   Employee e1; //creating an object of Employee
   Employee e2;
   return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;  //data member (also instance variable)
     string name;  //data member(also instance variable)
     float salary;
     Employee(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

Destructors

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
      Employee()
      {
         cout<<"Constructor Invoked"<<endl;
      }
      ~Employee()
      {
         cout<<"Destructor Invoked"<<endl;
      }
};
int main(void)
{
   Employee e1; //creating an object of Employee
   Employee e2; //creating an object of Employee
   return 0;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Another example explains the concept of destructor −

```
#include <iostream>

using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
```

```
Line::~Line(void) {
  cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

// Main function for the program
int main() {
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```

this Pointer

In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

1. It can be used to pass current object as a parameter to another function.
2. It can be used to refer current class data members if the member function variables and the data members of the class have same name.
3. It can be used to current object address.

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

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

```
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
      {
          this->id = id;
         this->name = name;
         this->salary = salary;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

Let us try the following example to understand the concept of this pointer −

```
#include <iostream>

using namespace std;

class Box {
  public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
```

```
    }
    int compare(Box box) {
      return this->Volume() > box.Volume();
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2)) {
    cout << "Box2 is smaller than Box1" <<endl;
  } else {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }

  return 0;
}
```

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

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
   friend data_type   function_name(argument/s);          // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.

- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

```
Length of box: 10
```

Example2:

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

class Box {
  double width;

  public:
    friend void printWidth( Box box );
```

```
    void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
  width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box ) {
  /* Because printWidth() is a friend of Box, it can
  directly access any member of this class */
  cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main() {
  Box box;

  // set box width without member function
  box.setWidth(10.0);

  // Use friend function to print the wdith.
  printWidth( box );

  return 0;
}
```

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

```
Width of box : 10
```

Arrays within a Class

- Arrays can be declared as the members of a class.
- The arrays can be declared as private, public or protected members of the class.A program to demonstrate the concept of arrays as class members

```
#include<iostream>
const int size=5;
class student
{
        int id;
        int marks[size];
        public:
        void getdata ();
        void totalmarks ();
} ;

void student :: getdata ()
{
        cout<<"\nEnter roll no: ";
        Cin>>id;
        for(int i=0; i<size; i++)
```

```
        {
                cout<<"Enter marks in subject"<<(i+1)<<": ";
                cin >>marks[i] ;
        }
}

void student :: totalmarks() //calculating total marks
{
        int total=0;
        for(int i=0; i<size; i++)
        {
                total  + = marks[i];
        }
        cout<<"\n\nTotal marks "<<total;
}

int  main()
{
        student stu;
        stu.getdata() ;
        stu.totalmarks() ;
        return  0
}
```

Output:

```
Enter roll no: 101
Enter marks in subject 1: 67
Enter marks in subject 2 : 54
Enter marks in subject 3 : 68
Enter marks in subject 4 : 72
Enter marks in subject 5 : 82
Total marks = 343
```

Array of Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

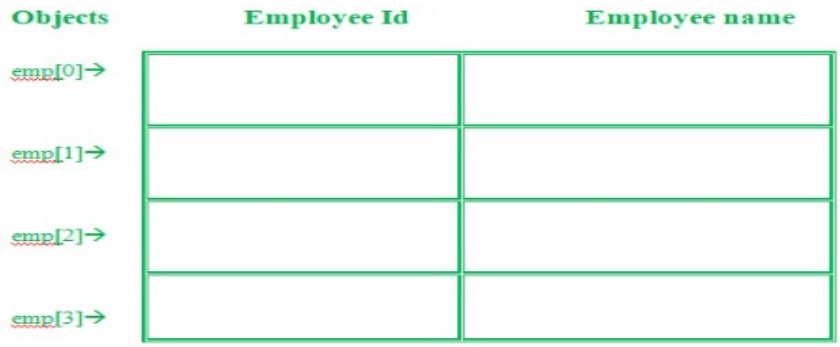
Syntax:

ClassName ObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

Example#1:

Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].



What if there is a requirement to add data of more than one Employee. Here comes the answer Array of Objects. An array of objects can be used if there is a need to store data of more than one employee. Below is the C++ program to implement the above approach-

```
// C++ program to implement
// the above approach
#include<iostream>
using namespace std;

class Employee
{
  int id;
  char name[30];
  public:

  // Declaration of function
  void getdata();

  // Declaration of function
  void putdata();
};

// Defining the function outside
// the class
void Employee::getdata()
{
  cout << "Enter Id : ";
  cin >> id;
  cout << "Enter Name : ";
  cin >> name;
}

// Defining the function outside
// the class
void Employee::putdata()
```

```
{
  cout << id << " ";
  cout << name << " ";
  cout << endl;
}

// Driver code
int main()
{
  // This is an array of objects having
  // maximum limit of 30 Employees
  Employee emp[30];
  int n, i;
  cout << "Enter Number of Employees - ";
  cin >> n;

  // Accessing the function
  for(i = 0; i < n; i++)
    emp[i].getdata();

  cout << "Employee Data - " << endl;

  // Accessing the function
  for(i = 0; i < n; i++)
    emp[i].putdata();
}
```

Output:

```
Enter Number of Employees - 3
Enter Id : 101
Enter Name : Mahesh
Enter Id : 102
Enter Name : Suresh
Enter Id : 103
Enter Name : Magesh
Employee Data -
101 Mahesh
102 Suresh
103 Magesh
```

Explanation:

In this example, more than one Employee's details with an Employee id and name can be stored.

- Employee emp[30] – This is an array of objects having a maximum limit of 30 Employees.
- Two for loops are being used-
    - First one to take the input from user by calling emp[i].getdata() function.
    - Second one to print the data of Employee by calling the function emp[i].putdata() function.

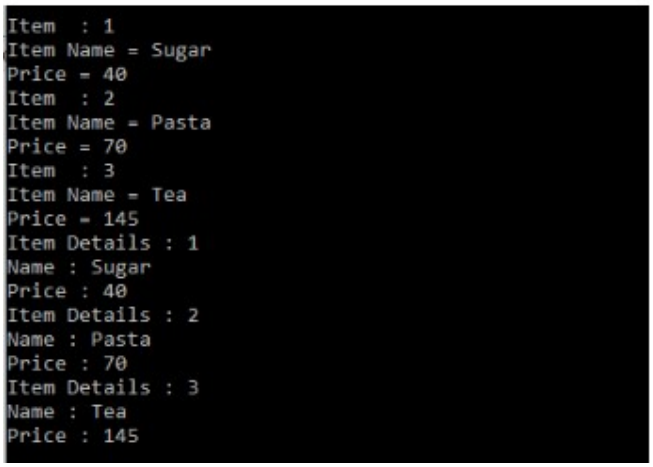
Example#2:

```
// C++ program to implement
// the above approach
```

```
#include<iostream>
using namespace std;
class item
{
  char name[30];
  int price;
  public:
  void getitem();
  void printitem();
};

// Function to get item details
void item::getitem()
{
  cout << "Item Name = ";
  cin >> name;
  cout << "Price = ";
  cin >> price;
}

// Function to print item
// details
void item ::printitem()
{
  cout << "Name : " << name <<
          "\n";
  cout << "Price : " << price <<
          "\n";
}

const int size = 3;

// Driver code
int main()
{
  item t[size];
  for(int i = 0; i < size; i++)
  {
    cout << "Item  : " <<
            (i + 1) << "\n";
    t[i].getitem();
  }

  for(int i = 0; i < size; i++)
  {
    cout << "Item Details : " <<
             (i + 1) << "\n";
    t[i].printitem();
  }
```

}

Output:



Advantages of Array of Objects:

1. The array of objects represent storing multiple objects in a single name.
2. In an array of objects, the data can be accessed randomly by using the index number.
3. Reduce the time and memory by storing the data in a single variable.

Module-3

C++ Overloading (Operator and Function)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- Methods/functions,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading

Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of parameters in the parameters list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types –

Example1:

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

class printData {
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};

int main(void) {
  printData pd;

  // Call print to print integer
  pd.print(5);

  // Call print to print float
  pd.print(500.263);

  // Call print to print character
  pd.print("Hello C++");

  return 0;
}
```

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

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Example2:

```
// Program of function overloading with different types of arguments.
    #include<iostream>
```

```
using namespace std;
int mul(int,int);
float mul(float,int);

int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Example3:

```
// program of function overloading when number of arguments vary.
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                                  //     class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

```
30
55
```

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
```

```
using namespace std;

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }

    // overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                    // apply negation
  D1.displayDistance();   // display D1

  -D2;                    // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

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

```
F: -11 I:-10
F: 5 I:-11
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

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

```
class Test
{
   int x;
    public:
   Test (int  i)
   {
     x=i;
   }
   void operator+( Test );
   void display();
};

void Test  :: operator+( Test  a)
{
   int m = x+a.x;
   cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
   Test  a1(5);
   Test  a2(4);
   a1+a2;
   return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

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

class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }
    void setLength( double len ) {
      length = len;
```

```
    }
    void setBreadth( double bre ) {
      breadth = bre;
    }
    void setHeight( double hei ) {
      height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};

// Main function for the program
int main() {
  Box Box1;               // Declare Box1 of type Box
  Box Box2;               // Declare Box2 of type Box
  Box Box3;               // Declare Box3 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	New	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

Sr.No	Operators & Example
1	Unary Operators Overloading
2	Binary Operators Overloading
3	Relational Operators Overloading
4	Input/Output Operators Overloading
5	++ and -- Operators Overloading
6	Assignment Operators Overloading
7	Function call () Operator Overloading
8	Subscripting [] Operator Overloading
9	Class Member Access Operator -> Overloading

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

Namespaces in C++

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

```
namespace namespace_name
{
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows −

```
name::code; // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions −

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

// first name space
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

int main () {
  // Calls function from first name space.
  first_space::func();
```

```
   // Calls function from second name space.
   second_space::func();

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Inside first_space
Inside second_space
```

The using directive

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code −

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

using namespace first_space;
int main () {
   // This calls function from first name space.
   func();

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Inside first_space
```

The 'using' directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows −

```
using  std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the **std** namespace will still need to be explicit as follows −

```
#include <iostream>
using std::cout;

int main () {
   cout << "std::endl is used with std!" << std::endl;
```

```
   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
std::endl is used with std!
```

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Nested Namespaces

Namespaces can be nested where you can define one namespace inside another name space as follows −

```
namespace   namespace_name1
{
        // code declarations
        namespace  namespace_name2
        {
                // code declarations
        }
}
```

You can access members of nested namespace by using resolution operators as follows −

```
// to access members of namespace_name2
using   namespace    namespace_name1::namespace_name2;

// to access members of namespace:name1
using   namespace   namespace_name1;
```

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows −

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }

   // second name space
   namespace second_space {
      void func() {
         cout << "Inside second_space" << endl;
      }
   }
}

using namespace first_space::second_space;
int main () {
   // This calls function from second name space.
```

```
   func();

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Inside second_space
```

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming

c