

Unit -1

Date: 25/2/2021

Reetu Bhardwaj

Namespace in C++

What is namespace?

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Syntax:

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

namespace abc

{

Int uid; // Data members

String name; // data members

Void printName()

{

Cout<<name<<endl;

}}

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don't have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.

Need of Namespace???

```
// A program to demonstrate need of namespace
int main()
{
    int value;
    value = 0;
    double value;
    value = 0.0;
}
```

Above program will give error due to same name variable in same scope

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

To call the namespace-enabled version of either function or variable, **scope resolution operator /prepend (::)** the namespace name as follows —

Name of namespace::code;

// code could be variable or function.

Example1

```
#include <iostream>

using namespace std;

namespace First {
    void sayHello() {
        cout<<"Hello First Namespace"<<endl;
    }
}

namespace Second {
    void sayHello() {
        cout<<"Hello Second Namespace"<<endl;
    }
}

int main()
{
    First::sayHello();
    Second::sayHello();
    return 0;
}
```

Example 2:

```
#include <iostream>

using namespace std;
namespace first
{
    int val = 500;
}
int val = 100;
```

```

int main()
{
    int val = 200;
    cout << first::val << '\n';
    return 0;
}

```

Example 3: (By using keyword)

// example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```

#include <iostream>

using namespace std;

namespace First{

    void sayHello(){

        cout << "Hello First Namespace" << endl;

    }

}

namespace Second{

    void sayHello(){

        cout << "Hello Second Namespace" << endl;

    }

}

using namespace Second;

int main () {

```

```
    sayHello();  
    return 0;  
}
```

Nested Namespace:

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

Syntax:

```
namespace abc  
{  
    //code declarations;  
    namespace xyz  
    {  
        // code declarations;  
    }  
}
```

How to access member of nested namespace (by using scope resolution operator)

// to access members of namespace_name2

```
using namespace abc::xyz;
```

// to access members of namespace:name1

```
using namespace namespace_name1;
```

Example:

```
#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;

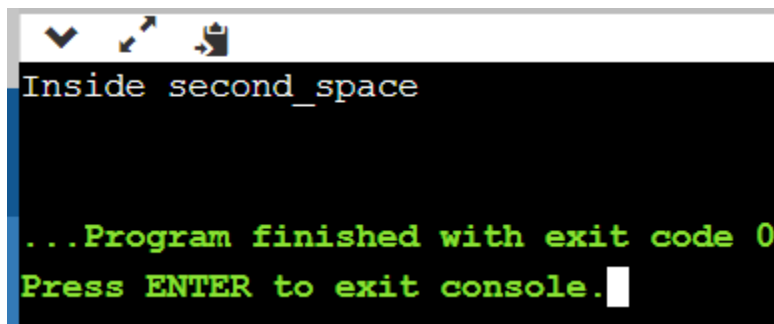
int main () {

    // This calls function from second name space.

    func();

    return 0;

}
```

A screenshot of a console window with a black background and green text. At the top, there are three small icons: a checkmark, a cursor, and a document. The text in the console reads: "Inside second_space", followed by "...Program finished with exit code 0", and "Press ENTER to exit console." with a white cursor at the end of the last line.

```
Inside second_space

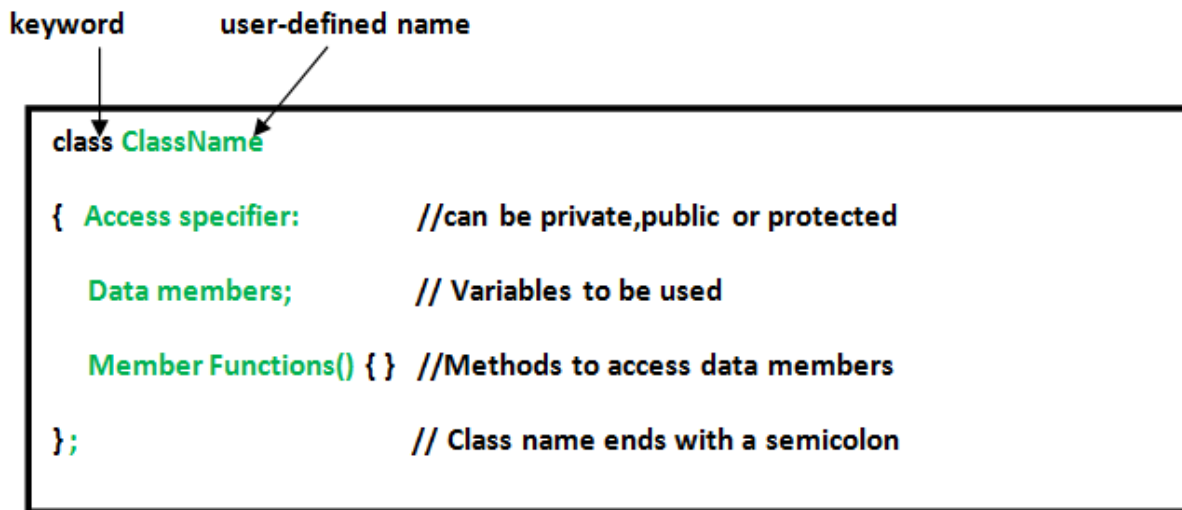
...Program finished with exit code 0
Press ENTER to exit console.
```

Defining Class and Creating Objects

Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

- ❖ A class definition starts with the keyword `class` followed by the class name; and the class body, enclosed by a pair of curly braces.
- ❖ A class definition must be followed either by a semicolon or a list of declarations.
- ❖ 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.

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.

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;

Accessing data members and member functions:

The data members and member functions of class can be accessed using the dot('.') operator with the object.

For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

Example

```
// C++ program to demonstrate
```

```
// accessing of data members
```

```
#include <iostream>
```

```
using namespace std;
```

```
class abc
```

```
{
```

```
    // Access specifier
```

```
public:
```

```
    // Data Members
```

```
    string name;
```

```
    // Member Functions()
```

```
    void printname()
```

```
{
```

```
    cout << "Name is: " << name;
```

```
}
```

```
};  
  
int main() {  
    // Declare an object of class abc  
    abc obj1;  
  
    // accessing data member  
    obj1.name = "Abhi";  
    //accessing member function  
    obj1.printname();  
    return 0;  
}
```

Output will be:

Name is Abhi

Access Specifiers/Modifiers in class

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as Data Hiding.

- However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated indirectly.
- The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.
- Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

Public

Private

Protected

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

Public:

- All the class members declared under the public specifier will be available to everyone.
- The public keyword is used to create public members (data and functions).
- The public members are accessible from any part of the program.
- The data members and member functions declared as public can be accessed by other classes and functions too.
- The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example:

```
class Patient {  
private:  
    int patientNumber;  
public:  
    string diagnosis;  
public:  
    void billing() {  
        // code  
    }  
    void makeAppointment() {  
        // code  
    }  
}
```

```
    }  
};
```

Example: 2

```
#include <iostream>  
  
using namespace std;  
  
// define a class  
  
class Sample {  
  
    // public elements  
  
    public:  
  
    int age;  
  
    void displayAge()  
  
    {  
  
        cout << "Age = " << age << endl;  
  
    }  
  
};  
  
int main() {  
  
    // declare a class object  
  
    Sample obj1;  
  
    cout << "Enter your age: ";  
  
    // store input in age of the obj1 object  
  
    cin >> obj1.age;  
  
    // call class function
```

```
    obj1.displayAge();  
    return 0;  
}
```

Example 2:

```
#include<iostream>  
  
using namespace std;  
  
// class definition  
class Circle  
{  
    public:  
        double radius;  
        double compute_area()  
        {  
            return 3.14*radius*radius;  
        }  
};  
  
// main function  
int main()  
{  
    Circle obj;  
  
    // accessing public data member outside class
```

```
obj.radius = 5.5;

cout << "Radius is: " << obj.radius << "\n";

cout << "Area is: " << obj.compute_area();

return 0;

}
```

Private Access Specifiers

- The class members declared as private can be accessed only by the member functions inside the class.
- They are not allowed to be accessed directly by any object or function outside the class.
- Only the member functions or the friend functions are allowed to access the private data members of a class.

Example: 1

```
#include<iostream>

using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
```

```

double compute_area()
{
    // member function can access private
    // data member radius

    return 3.14*radius*radius;
}

};

// main function

int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class

    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();

    return 0;
}

```

The output of above program is a compile time error because we are not allowed to access the private data members of a class directly outside the class. Yet an access to obj.radius is attempted, radius being a private data member we obtain a compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

Modified example:

```
#include<iostream>

using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        void compute_area(double r)
        { // member function can access private
            // data member radius

            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;

            cout << "Area is: " << area;

        }
}
```

```
};

int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);

    return 0;
}
```

Example 2:

```
#include<iostream>
using namespace std;
class emp
{
    private:
    int id;
    char name[20];
    float sal;
    void getemp()
    {
```

```

        cout<<"Enter emp id, name, salary"<<endl;

        cin>>id>>name>>sal;
    }

    public:

    void putemp()
    {
        getemp();

        cout<<id<<name<<sal;

    }

};

int main()
{ emp e;

e.putemp();

return 0;

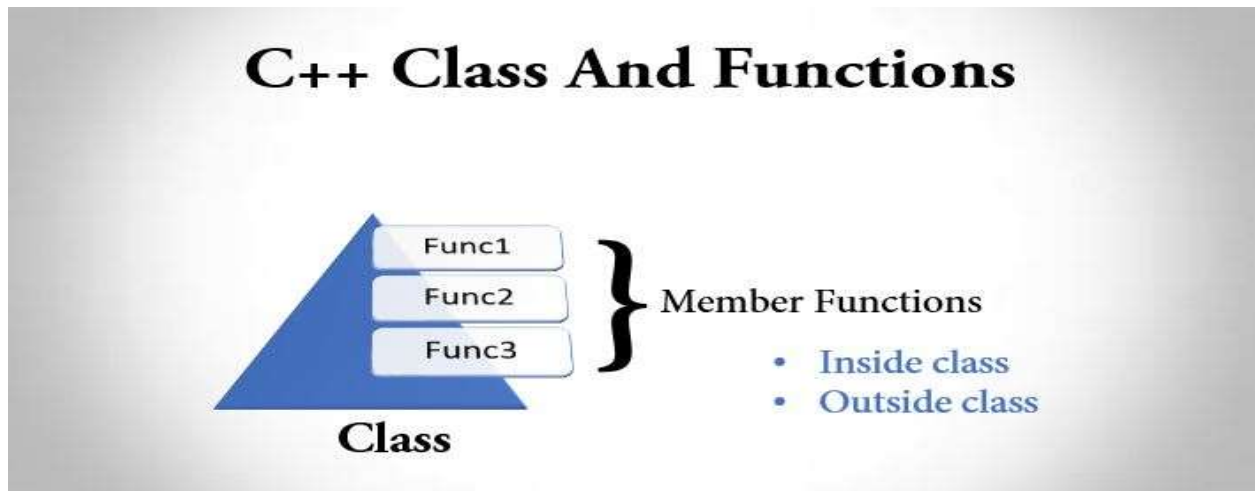
}

```

Protected:

- Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of it's class unless with the help of friend class.
- The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Defining functions in class:



The functions associated with a class are called member functions of that class.

Member functions must be declared inside the class but they can be defined either inside the class or outside the class.

Different ways of defining member functions of a class

There are two ways in which the member functions can be defined :

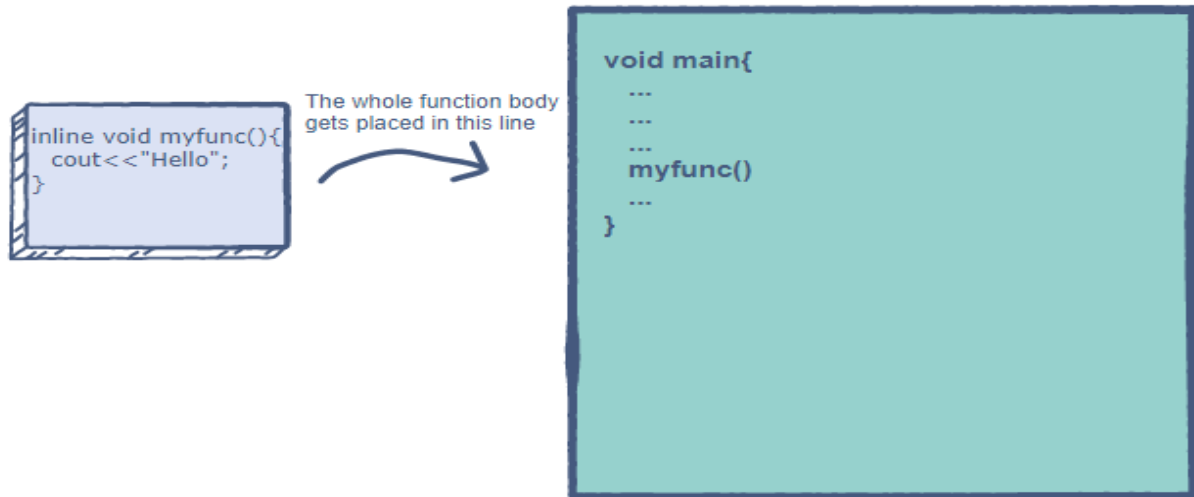
- **Inside the class definition**
- **Outside the class definition**

1. Inside the class definition:

As the name suggests, here the functions are defined inside the class.

Functions defined inside the class are treated as inline functions automatically if the function definition doesn't contain looping statements or complex multiple line operations.

Inline function???



```
Void myFunc()
{
Cout<<"Hello";

}

myFunc() //callee
```

An inline function is a function that is expanded inline when it is invoked, thus saving time.

The compiler replaces the function call with the corresponding function code, which reduces the overhead of function calls

When an inline function is called, the compiler replaces all the calling statements with the function definition at run-time.

Every time an inline function is called, the compiler generates a copy of the function's code, in place, to avoid the function call.

Now let's back to the topic.....

Consider the following syntax

```
class class_name
{
    private:
        declarations;
    public:
        function_declaration(parameters)
        {
            function_body;
        }
};
```

Example:

```
#include <iostream>

using namespace std;
```

```
class Example
{
    private:
        int val;
    public:
        //function to assign value
        void init_val(int v)
        {
            val=v;
        }
        //function to print value
        void print_val()
        {
            cout<<"val: "<<val<<endl;
        }
};
```

```
int main()
{
    //create object
    Example Ex;
    Ex.init_val(100); //
```

```
Ex.val= 100  
  
return 0;  
  
}
```

In the above example, public member functions `init_val()` and `print_val()` are defined inside the class definition

2. Defining member function outside of the class definition

As the name suggests, here the functions are defined outside the class however they are declared inside the class.

Functions should be declared inside the class to bound it to the class and indicate it as it's member but they can be defined outside of the class.

To define a function outside of a class, scope resolution operator `::` is used.

Syntax for declaring function outside of class

```
class class_name abc  
{  
.....  
.....  
public:  
    void add(int a,int b); //function declaration  
};  
  
//function definition outside class
```



```
void abc:: add(int a, int b)

{

    .....; // function definition }
```

Example 1:

```
#include <iostream>

using namespace std;

class car
{
    private:

        int car_number;

        char car_model[10];

    public:

        void getdata(); //function declaration

        void showdata();

};

// function definition

void car::getdata()

{

    cout<<"Enter car number: "; cin>>car_number;

    cout<<"\n Enter car model: "; cin>>car_model;

}

void car::showdata()
```

```
{  
    cout<<"Car number is "<<car_number;  
    cout<<"\n Car model is "<<car_model;  
}  
  
// main function starts  
  
int main()  
{  
    car c1;  
    c1.getdata();  
    c1.showdata();  
    return 0;  
}
```

Here is this program, the functions showdata() and getdata() are declared inside the class and defined outside the class. This is achieved by using scope resolution operator

Types of Member Function in C++

1. Simple Function
2. Inline Function
3. Static Function
4. Friend Function

Simple Function:

We have already discussed about simple function as class examples

Inline Function:

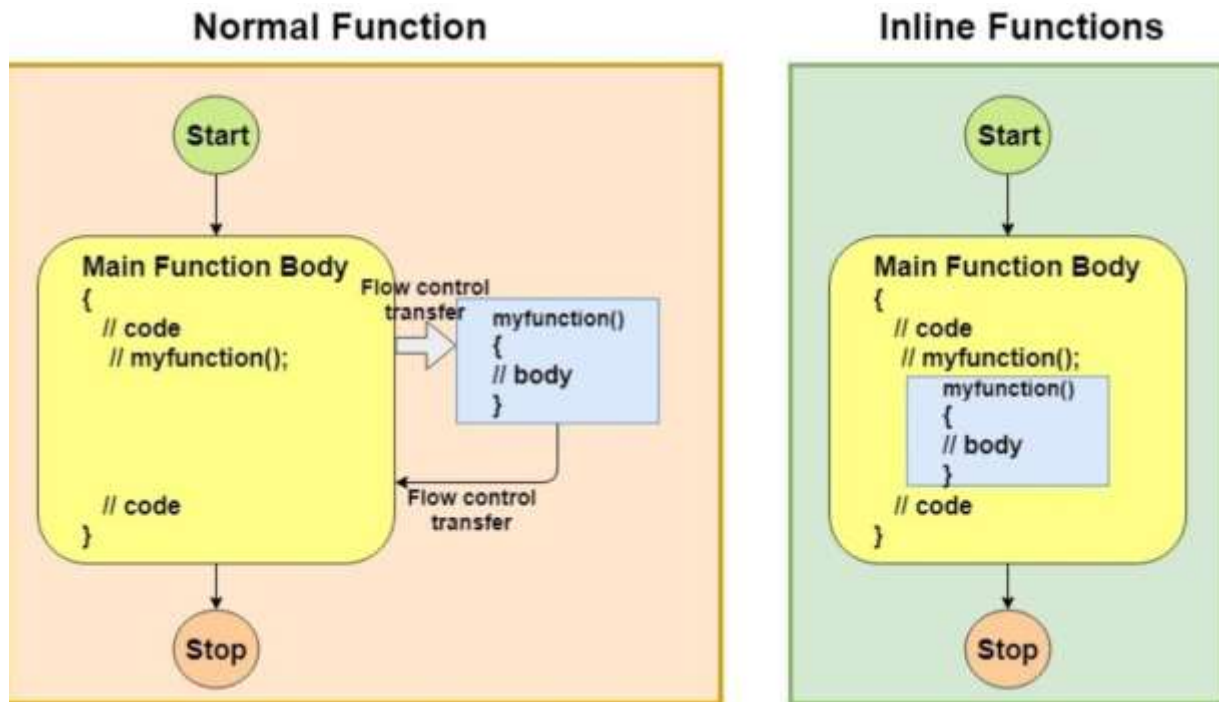
When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function.

- ❖ The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function.
- ❖ This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee)
- ❖ This overhead occurs for small functions because execution time of small function is less than the switching time

- ❖ Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
- ❖ This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```



Compiler may not perform in-lining in such circumstances like:

- 1) If a function contains a loop. (for, while, do-while)
- 2) If a function contains static variables.
- 3) If a function is recursive.
- 4) If a function contains switch or goto statement.

Advantage/Disadvantage:

- 1) Function call overhead doesn't occur.
- 2) It also saves the overhead of push/pop variables on the stack when function is called.
- 3) It also saves overhead of a return call from a function.
- 4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function.

Disadvantage:

- 1) The added variables from the in-lined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization
- 2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

Example: 1

```
#include <iostream>

using namespace std;

inline void displayNum(int num) {

    cout << num << endl;

}

int main() {

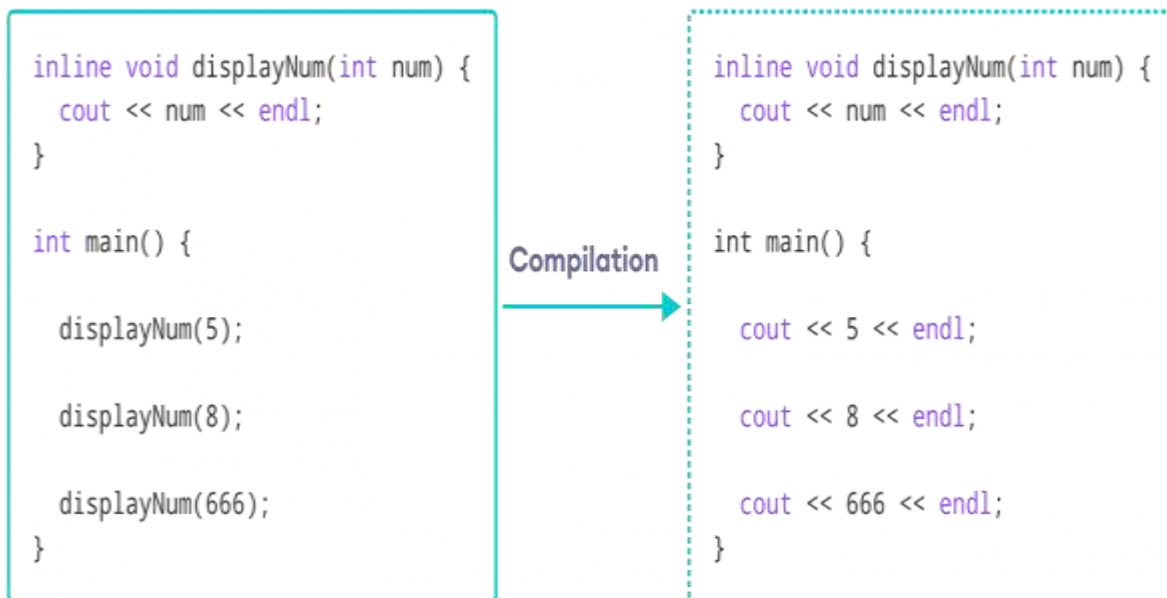
    displayNum(5);

    displayNum(8);

    displayNum(666);

    return 0;

}
```



Inline function and classes:

It is also possible to define the inline function inside the class.

In fact, all the functions defined inside the class are implicitly inline.

Thus, all the restrictions of inline functions are also applied here.

Syntax:1

```
class S
{
public:
    int square(int s)
    {
        // this function is automatically inline
        // function body
    }
};
```

Syntax 2:

```
class S {
public:
    int square(int s); // declare the function
```



```
};
```

```
inline int S::square(int s) // use inline prefix
```

```
{ }
```

Example:

```
#include <iostream>
```

```
using namespace std;
```

```
class operation
```

```
{
```

```
    int a,b,add,sub,mul;
```

```
    float div;
```

```
public:
```

```
    void get();
```

```
    void sum();
```

```
    void difference();
```

```
    void product();
```

```
    void division();
```

```
};
```

```
inline void operation :: get()
```

```
{
```

```
    cout << "Enter first value:";
```

```
    cin >> a;
```

```
    cout << "Enter second value:";

    cin >> b;

}
```

```
inline void operation :: sum()

{

    add = a+b;

    cout << "Addition of two numbers: " << a+b << "\n";

}
```

```
inline void operation :: difference()

{

    sub = a-b;

    cout << "Difference of two numbers: " << a-b << "\n";

}
```

```
inline void operation :: product()

{

    mul = a*b;

    cout << "Product of two numbers: " << a*b << "\n";

}
```

```
inline void operation ::division()
```

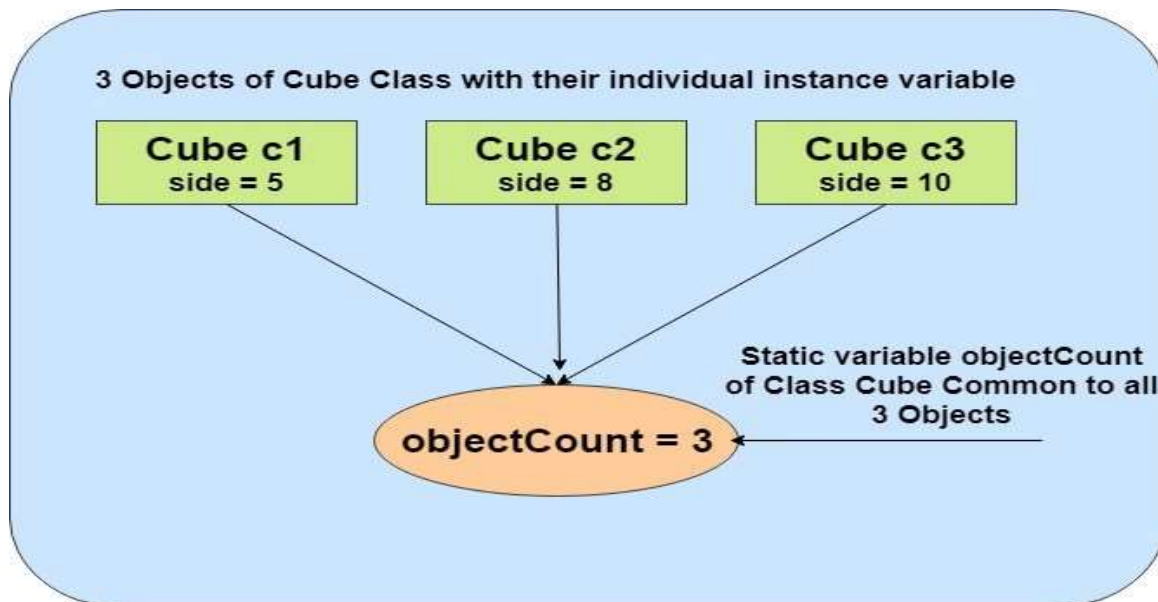
```
{  
    div=a/b;  
    cout<<"Division of two numbers: "<<a/b<<"\n" ;  
}  
  
int main()  
{  
    cout << "Program using inline function\n";  
    operation s;  
    s.get();  
    s.sum();  
    s.difference();  
    s.product();  
    s.division();  
    return 0;  
}
```

Static Data Members....

Static is something that holds its position.

Static is a keyword which can be used with data members as well as the member functions.

- ❖ Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- ❖ It is visible only within the class, but its lifetime is the entire program.
- ❖ A static variable is normally used to maintain value common to the entire class. For e.g, to hold the count of objects created.



Note that the type and scope of each static member variable must be declared outside the class definition.

member function by itself, using class name and scope resolution :: operator.

. Syntax of Declaring/Defining Static data members

Declaration:

`static data_type member_name;`

`static int a;`

Defining Static data member:

It should be defined outside of the class following this syntax:

`data_type class_name :: member_name =value;`

Example 1:

```
#include <iostream>
```

```
using namespace std;
```

```
class Demo
```

```
{
```

```
    public:
```

```
        static int ABC; //declaration
```

```
};
```

```
//defining
```

```
int Demo :: ABC =10;
```

```
int main()
```

```
{
```

```
    cout<<"\nValue of ABC: "<<Demo::ABC;

    return 0;

}
```

Example 2:

```
#include <iostream>

using namespace std;

class Account {

public:

    int accno; //data member (also instance variable)

    string name;

    static int count;

    Account(int a, string n)

    {

        accno = a;

        name = n;

        count++;

    }

    void display()

    {

        cout<<accno<<" "<<name<<endl;
```

```

    }
};

int Account::count=0;

int main(void) {

    Account a1 =Account(201, "Sanjay"); //creating an object of Account

    Account a2=Account(202, "Nakul");

    Account a3=Account(203, "Ranjana");

    a1.display();

    a2.display();

    a3.display();

    cout<<"Total Objects are: "<<Account::count;

    return 0;

}

```

Static Member Functions:

A static member function is a special member function, which is used to access only static data members, any other normal data member cannot be accessed through static member function.

- Just like static data member, static member function is also a class function; it is not associated with any class object.
- These functions work for the class as whole rather than for a particular object of a class.

- It can be called using an object and the direct member access (.) Operator. But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.
- These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

We can access a static member function with class name, by using following syntax:

class_name :: function_name (parameter);

Example:

```
#include <iostream>

using namespace std;

class Demo
{
    private:
        //static data members
        static int X;
        static int Y;
    public:
        //static member function
        static void Print()
        {
```



```
        cout <<"Value of X: " << X << endl;

        cout <<"Value of Y: " << Y << endl;

    }

};

//static data members initializations

int Demo :: X =10;

int Demo :: Y =20;

int main()

{

    Demo OB;

    //accessing class name with object name

    cout<<"Printing through object name:"<<endl;

    OB.Print();

    //accessing class name with class name

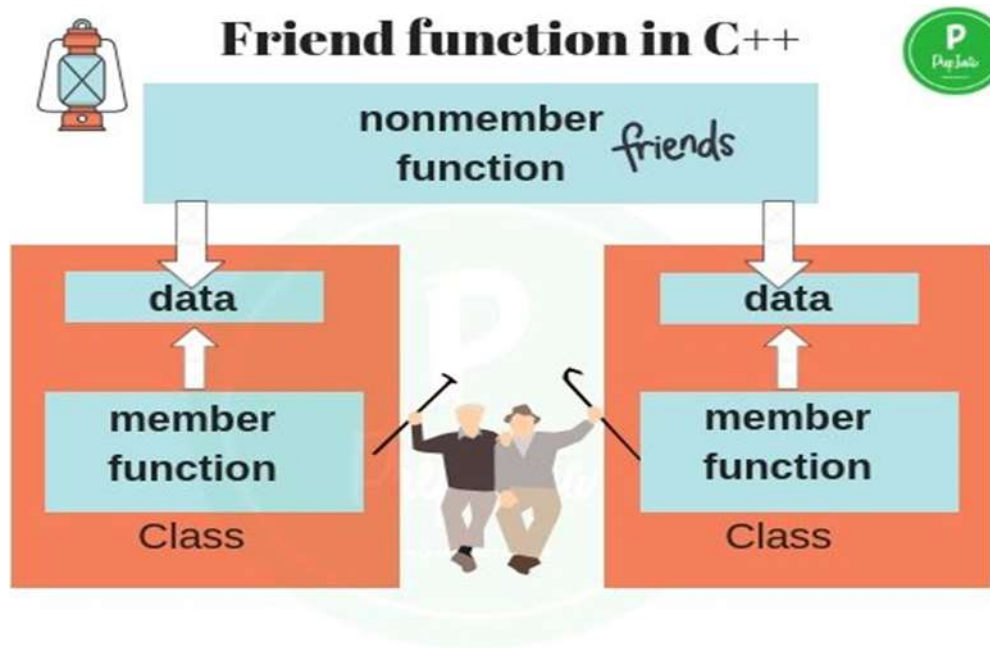
    cout<<"Printing through class name:"<<endl;

    Demo::Print();

    return 0;

}
```

FRIEND FUNCTION in C++



Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside.

However, there is a feature in C++ called friend functions that break this rule and allow us to access member functions from outside the class.

- 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.

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
- Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

Declaration of friend function in C++

```
class
{

//data members;
//member function

friend data_type function_name(argument/s);

};
```

Note: 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.

Example 1:

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;
}

```

Example 2:

WAP to Add Members of Two Different Classes

// Add members of two different classes using friend functions

```
#include <iostream>
```

```
using namespace std;
```

// forward declaration

```
class ClassB;
```

```
class ClassA {
```

public:

// constructor to initialize numA to 12

ClassA() : numA(12) {}

private:

int numA;

// friend function declaration

friend int add(ClassA, ClassB);

};

class ClassB {

public:

// constructor to initialize numB to 1

ClassB() : numB(1) {}

private:

int numB;

// friend function declaration

friend int add(ClassA, ClassB);

};

// access members of both classes

int add(ClassA objectA, ClassB objectB) {

return (objectA.numA + objectB.numB);

```
}
```

```
int main() {  
    ClassA objectA;  
    ClassB objectB;  
    cout << "Sum: " << add(objectA, objectB);  
    return 0;  
}
```

Friend class in c++

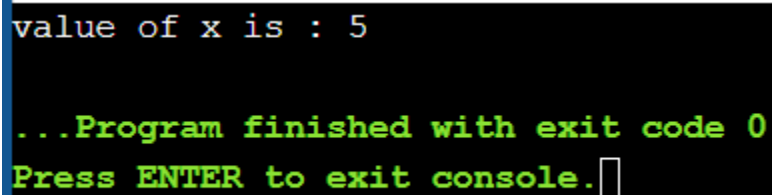
A friend class is a class that can access the private and protected members of a class in which it is declared as friend.

- This is needed when we want to allow a particular class to access the private and protected members of a class.
- When a class is declared a friend class, all the member functions of the friend class become friend functions.
- A friend class can access both private and protected members of the class in which it has been declared as friend

Example 1:

```
#include <iostream>
```

```
using namespace std;
class A
{
    int x = 5;
    friend class B;    // friend class.
};
class B
{
public:
    void display(A &a)
    {
        cout << "value of x is : " << a.x;
    }
};
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

A screenshot of a console window with a black background. The first line of output is "value of x is : 5" in a light blue/cyan monospace font. The second line is "...Program finished with exit code 0" in a green monospace font. The third line is "Press ENTER to exit console." in a green monospace font, followed by a white cursor icon (a small square) at the end of the line.

```
value of x is : 5
...Program finished with exit code 0
Press ENTER to exit console.█
```


In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A

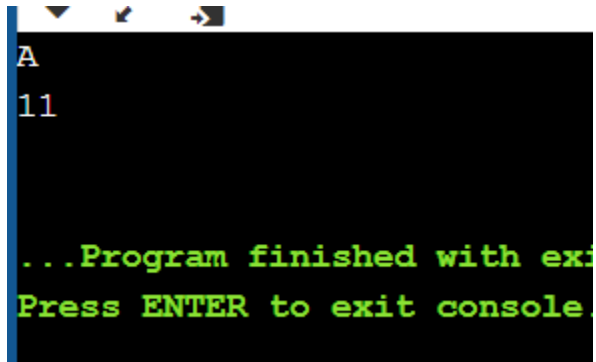
Example 2:

```
#include <iostream>

using namespace std;
class XYZ {
private:
    char ch='A';
    int num = 11;
public:
    /* This statement would make class ABC
    * a friend class of XYZ, this means that
    * ABC can access the private and protected
    * members of XYZ class.
    */
    friend class ABC;
};
class ABC {
public:
    void disp(XYZ obj){
        cout<<obj.ch<<endl;
        cout<<obj.num<<endl;
    }
};
int main() {
    ABC obj;
    XYZ obj2;
```

```
    obj.disp(obj2);  
    return 0;  
}
```

Output:

A screenshot of a console window with a black background and a blue title bar. The title bar contains three icons: a downward arrow, a leftward arrow, and a rightward arrow. The console output is as follows:
A
11

...Program finished with exit code 0
Press ENTER to exit console.
The text is displayed in a monospaced font, with the first two lines in white and the last two lines in green.

Object as Function argument and returning objects to function:

In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so.

1. Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

`function_name(object_name);`

Example :

```
// C++ program to calculate the average marks of two
students

#include <iostream>
using namespace std;

class Student {

public:
    double marks;

    // constructor to initialize marks
    Student(double m) {
        marks = m;
    }
}
```

```

    }
};

// function that has objects as parameters
void calculateAverage(Student s1, Student s2) {

    // calculate the average of marks of s1 and s2
    double average = (s1.marks + s2.marks) / 2;

    cout << "Average Marks = " << average << endl;

}

int main() {
    Student student1(88.0), student2(56.0);

    // pass the objects as arguments
    calculateAverage(student1, student2);

    return 0;
}

```

Here, we have passed two Student objects student1 and student2 as arguments to the calculateAverage() function.

```


#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ..
    calculateAverage(student1, student2);
    ... ..
}

```



2. Returning Object as argument

Syntax:

`object = return object_name;`

Example:

```
#include <iostream>
using namespace std;
class Student {
    public:
        double marks1, marks2;
};

// function that returns object of Student
Student createStudent() {
    Student student;

    // Initialize member variables of Student
    student.marks1 = 96.5;
    student.marks2 = 75.0;

    // print member variables of Student
    cout << "Marks 1 = " << student.marks1 << endl;
    cout << "Marks 2 = " << student.marks2 << endl;

    return student;
}

int main() {
    Student student1;

    // Call function
    student1 = createStudent();
}
```

```
    return 0;  
}
```

Output

```
Marks1 = 96.5  
Marks2 = 75
```

```
#include<iostream>  
  
class Student {...};  
  
Student createStudent() {  
    Student student;  
    ... ..  
    return student;  
}  
  
int main() {  
    ... ..  
    student1 = createStudent();  
    ... ..  
}
```

function call

In this program, we have created a function `createStudent()` that returns an object of `Student` class.

We have called `createStudent()` from the `main()` method.

```
// Call function  
student1 = createStudent();
```

Here, we are storing the object returned by the `createStudent()` method in the `student1`