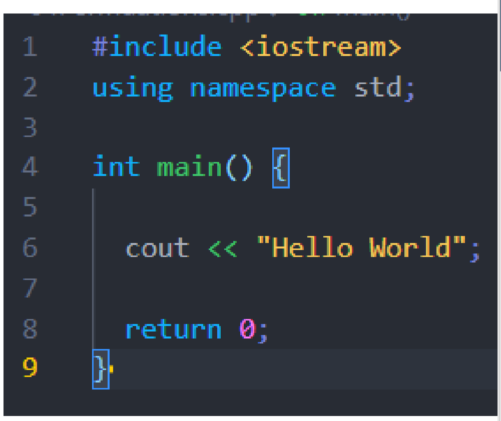
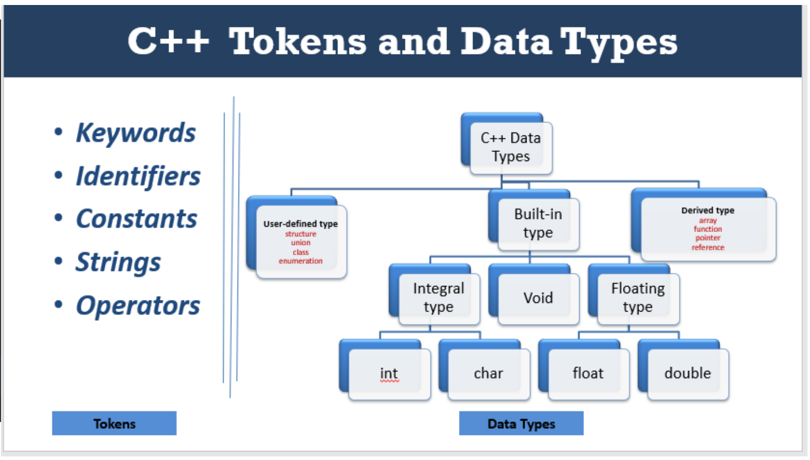
Q1. What is C++ Programming? Write a basic C++ code? Name some C++ Tokens and Data Types?

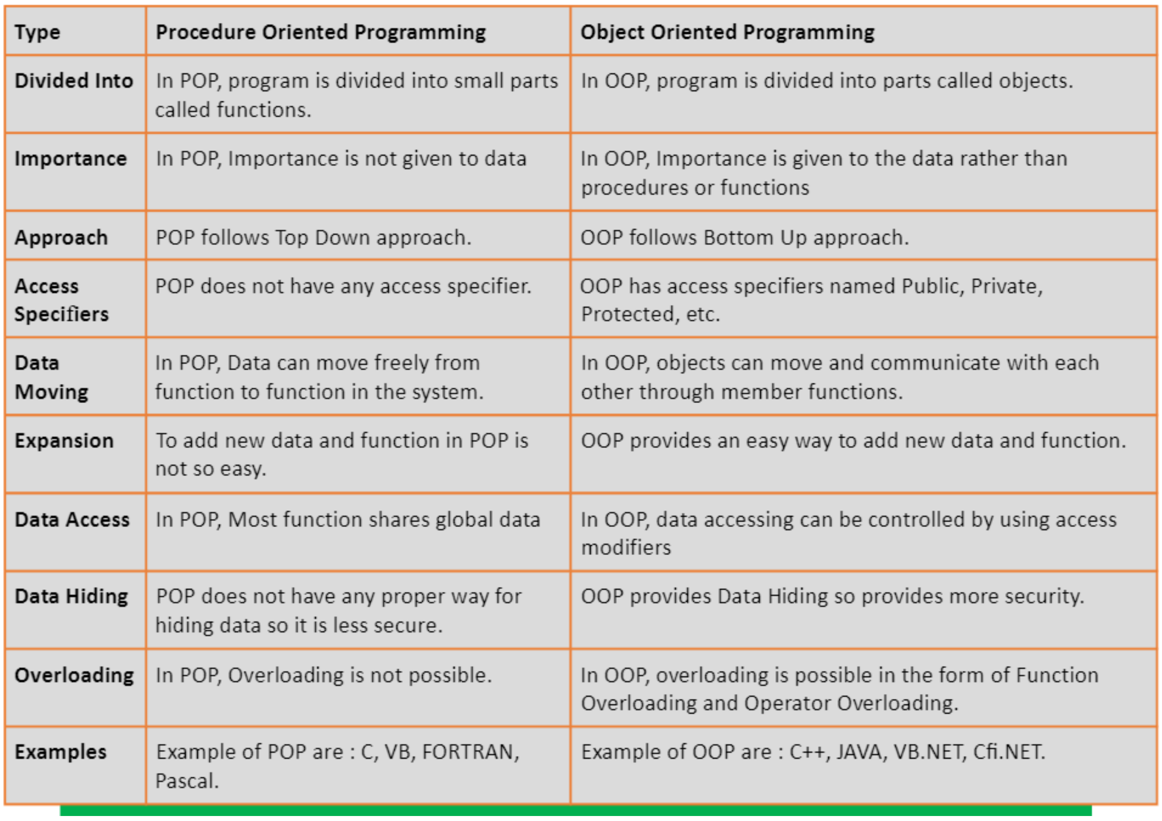
--- C++ is an extension of the C programming language. It’s a powerful, high-level programming language that is widely used for system/software development, game programming, and performance-critical applica tions.



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

int main() {

    cout << "Hello World";

    return 0;
}
```



Q2. Difference Between POP vs OOP?



Type	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data	In OOP, Importance is given to the data rather than procedures or functions
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function shares global data	In OOP, data accessing can be controlled by using access modifiers
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, Cfi.NET.

Q3. What is OOP? Explain the major features of OOP.

Object-Oriented Programming (OOP) is a way of designing and organizing software that focuses on using objects to represent real-world entities. In simpler terms, OOP allows programmers to create models of things in the world (like cars, animals, or people) and define how these things interact with each other.

Key Features of OOP:

1. **Class:** A class is like a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have. For example, a Dog class might have attributes like breed and age, and methods like bark() and fetch().

2. **Object:** An object is a specific instance of a class. It represents a particular entity with its own unique data. For instance, if Dog is a class, then a specific dog named "Buddy" that is a Golden Retriever is an object of that class. Each object can have different values for its attributes.

3. **Encapsulation:** Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

4. **Abstraction:** Abstraction is a process of hiding the implementation details and showing only functionality to the user. It allows programmers to work with higher-level concepts without needing to understand all the underlying complexities.

5. **Inheritance:** Inheritance is the process by which one object can acquire the properties of another. This promotes code reuse and establishes a relationship between classes. For example, a base class Animal, one can create derived classes like Dog and Cat that inherit common characteristics from Animal, such as eat() and sleep() methods.

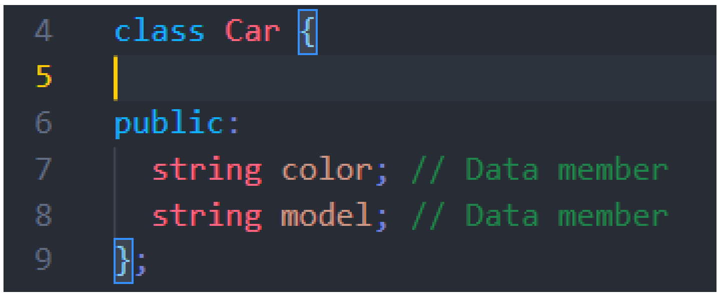
6. **Polymorphism:** Polymorphism enables objects to be treated as instances of their parent class, allowing methods to be used in different ways depending on the object. This can be achieved through method overriding (where a subclass provides a specific implementation of a method) or method overloading (where multiple methods have the same name but different parameters). For example, a method makeSound() could behave differently for a Dog (barking) and a Cat (meowing).

Q4. What is class or object in C++? Describe the components of a class, such as data members, member function and access specifier. Give a basic syntax example of a **class** or **object** in C++.

Class: A class is like a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have.

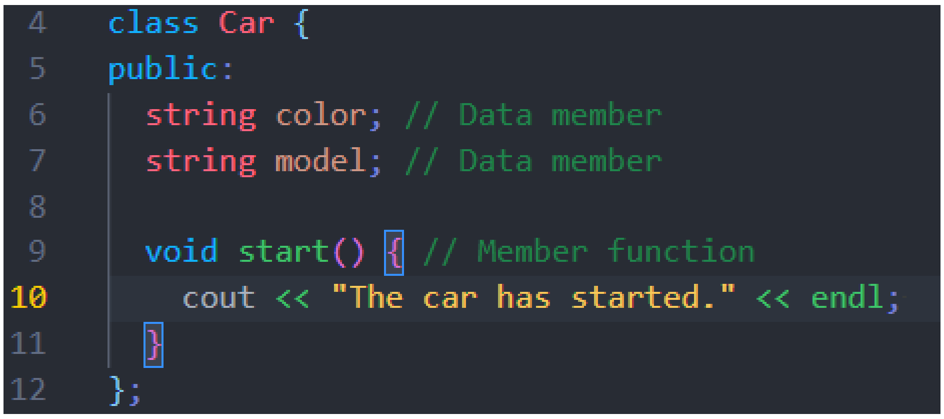
Object: An object is a specific instance of a class. It represents a particular entity with its own unique data.

Data Members: A data member is a variable that is declared inside a class. It represents the attributes or properties of an object. For example, in a Car class, there might have data members like color and model.



```cpp
class Car {

public:
  string color; // Data member
  string model; // Data member
};
```

Member Functions: A member function is a function that is defined within a class and operates on the data members of that class. It can manipulate the data members and perform actions related to the object. For example, a start() function in the Car class could be a member function.



```cpp
class Car {
public:
  string color; // Data member
  string model; // Data member

  void start() { // Member function
    cout << "The car has started." << endl;
  }
};
```

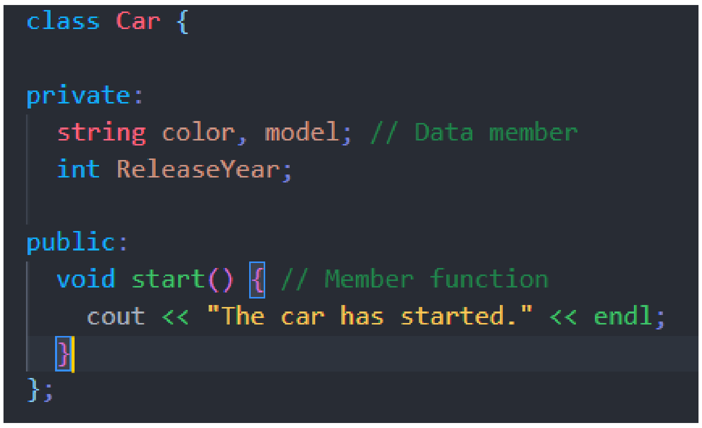
Access Specifier: Access specifiers are keywords that determine the accessibility of class members (data members and member functions). The three main access specifiers in C++ are **Public**, **Private**, and **Protected**.

Private: Accessible only to member functions of the class. The default access for class members is private.

Public: Accessible both by other members of the class and other part of the program that contains the class.

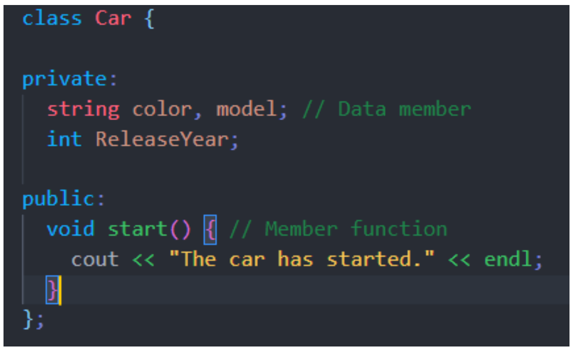
Protected: Data members and functions are available to derived classes only.

Syntax of class:

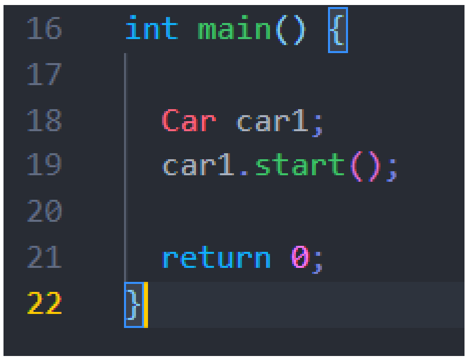


```
class Car {

private:
  string color, model; // Data member
  int ReleaseYear;

public:
  void start() { // Member function
    cout << "The car has started." << endl;
  }
};
```

Syntax of Object:



```
class Car {

private:
  string color, model; // Data member
  int ReleaseYear;

public:
  void start() { // Member function
    cout << "The car has started." << endl;
  }
};
```



```
int main() {

  Car car1;
  car1.start();

  return 0;
}
```

Q5. Explain Object initialization with example.

Object can be initialized in 3 ways. Here:

1. By Assignment
2. By Public Member Functions
3. Constructor

1. **By Assignment:** Only work for public data members. No control over the operations on data members

```
#include <iostream>
using namespace std;
  class circle
  {
      public:
        double radius;
  };

int main()
{
   circle c1;                // Declare an instance of the class circle
   c1.radius = 5;   // Initialize by assignment
}
```

2. **By Public Member Functions:**

```
#include <iostream>
using namespace std;
    class circle
    {
      private:
            double radius;
      public:
            void set (double r)
                 {radius = r;}
            double get_r ()
                 {return radius;}
    };
```

```
int main() {
    circle c;                    // an object of circle class
    c.set(5.0);                  // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
              // access a private data member with an accessor
}
```

3. By Constructor **:**

- They are publicly accessible
- Have the same name as the class
- There is no return type
- Are used to initialize class data members
- They have different signatures

```
 #include <iostream>
 using namespace std;
     class Rectangle
     {
             private:
               int width;
               int length;
             public:
               Rectangle();
               Rectangle(const Rectangle &r);
               Rectangle(int w, int l);
               void set(int w, int l);
               int area();
     }

int main()
{
   Rectangle r1, r2(5), r3(60,80), r4.set(20,10);   // Declare an instance of the
initialization of object

}
```

Q6. Difference between structure and class.

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.
Declaration	*struct structure_name{ type struct_member 1; type struct_member 2; type struct_member 3; . type struct_memberN; };*	*class class_name{ data member; member function; };*
Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.
Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

Q7.What is inline function? Write a basic syntax of inline-function.

An inline function is a special type of function in C++ that is defined with the **inline** keyword. Instead of performing a regular function call, the compiler tries to replace the function call with the actual code of the function. This can make the program run faster, especially for small functions that are called frequently.

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

inline int Max(int x, int y) {

  return (x > y)? x : y;

}

int main( ) {

  cout << "Max (20,10): " << Max(20,10) << endl;

  cout << "Max (0,200): " << Max(0,200) << endl;

  cout << "Max (100,1010): " << Max(100,1010) << endl;

  return 0;

}
```

Q8.Define categories of user-defined function.

Ans--- There are four types of user-defined function. Here:

1. Function with no argument and no return value.
2. Function with no argument but return value.
3. Function with argument but no return value.
4. Function with argument and return value.

1. Function with no argument and no return value.

```
void MyPrint(){
        cout << "Printing from a function.\n";
}
int main(){
        MyPrint();
        return 0;
}
```

2. Function with no argument but return value.

```
int Add() {

     int a = 10, b = 5;

     return a+b;

}

int main(){

          cout << Add();
          return 0;
}
```

3. Function with argument but no return value.

```
Void Add(int a, int b) {

     cout <<  a+b;

}

int main(){

          Add();
          return 0;
}
```

5. Function with argument and return value.

```
Void Add(int a, int b) {

     cout <<  a+b;

}

int main(){

          Add();
          return 0;
}
```

Q9. Define function overloading with example.

Answer--- :

Function overloading is a feature in C++ that allows you to create multiple functions with the same name but different parameters. This means you can have several functions that perform similar tasks but accept different types or numbers of arguments. The compiler determines which function to call based on the arguments provided.

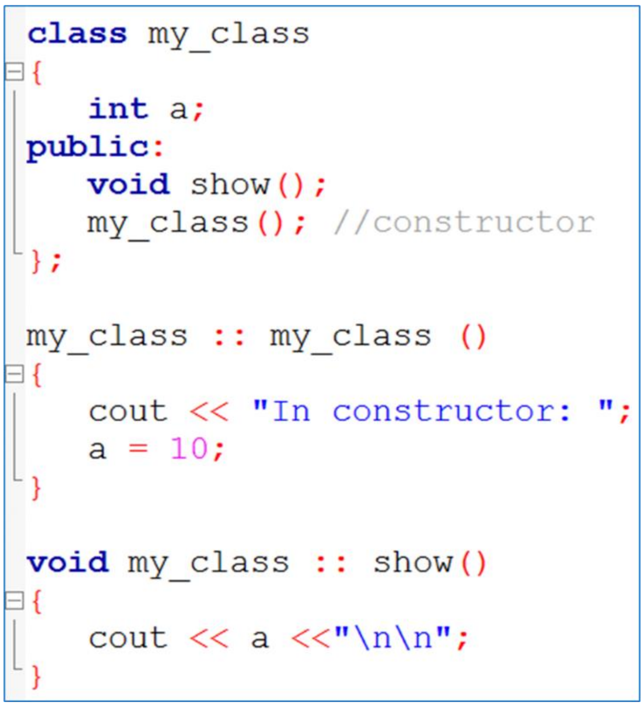
```
#include <iostream>
using namespace std;


int add(int a, int b) {
    return a + b;
}
int add(int a, int b, int c) {
    return a + b + c;
}
double add(double a, double b) {
    return a + b;
}
int main() {
    cout << "Sum of 2 and 3: " << add(2, 3) << endl;            // Calls the first function
    cout << "Sum of 2, 3 and 4: " << add(2, 3, 4) << endl;    // Calls the second function
    cout << "Sum of 2.5 and 3.5: " << add(2.5, 3.5) << endl;  // Calls the third function
    return 0;
}
```

Q10. What is constructor? Define characteristics of constructor. Write a basic syntax of constructor.

Answer---:

A constructor function is a **special** member function that has the **same name as the class** of which it is a part and **called(invoked) each time** an object of that class is created.

- Declared in the public section.
- Invoked automatically when the objects are created.
- Do not have return type.
- Can have default arguments.
- Cannot be virtual.
- Cannot be refered to their addresses.
- They make 'implicit calls' to the operators 'new' and 'delete' when memory allocation is required.



```
class my_class
{
    int a;
public:
    void show();
    my_class(); //constructor
};

my_class :: my_class ()
{
    cout << "In constructor: ";
    a = 10;
}

void my_class :: show()
{
    cout << a <<"\n\n";
}
```

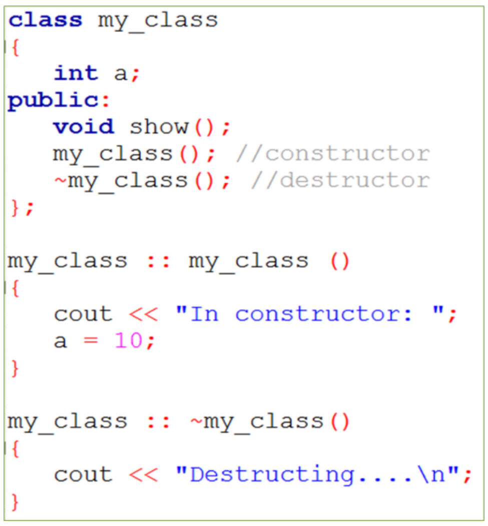
Q11. What is Destructor? Purposes of destructor. Write a basic syntax of destructor.

Answer---:

A destructor is a special member function in C++ that is automatically invoked when an object of a class is destroyed. This typically occurs when the object goes out of scope or is explicitly deleted using the delete operator. The primary role of a destructor is to perform cleanup tasks, such as releasing resources that were allocated during the object's lifetime.

Purposes:

- Constructor allocates memory to the object. This allocated memory must be de-allocated before the object is destroyed. This job of memory de-allocation from object is done by special member function of the class. This member function is destructor.
- The main use of destructors is to release dynamic allocated memory and perform cleanup.
- Destructors are automatically called when an object is destroyed.
- A destructor will have exact same name as the class prefixed with a tilde (~)
- A destructor takes no arguments and has no return type.



```
class my_class
{
    int a;
public:
    void show();
    my_class(); //constructor
    ~my_class(); //destructor
};

my_class :: my_class ()
{
    cout << "In constructor: ";
    a = 10;
}

my_class :: ~my_class()
{
    cout << "Destructing....\n";
}
```

Q12.Define Parameterized Constructor with syntax example.

Answer---:

C++ permits us to achieve the objects but passing argument to the constructor function when the object are created. The constructor that can take arguments are called parameterized constructors.

```
class abc{
int m, n;
public:
   abc(int x, int y); //parameterized constructor

   m = x;
   n = y;
};
```

Q13.Define constructor overloading with example.

Answer---:

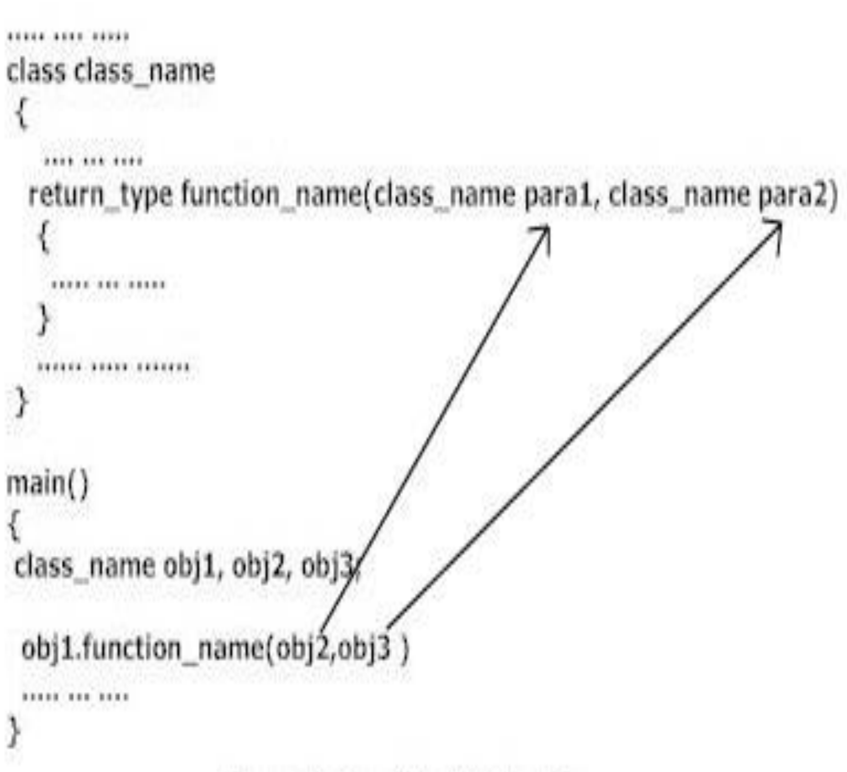
Constructors Overloading are used to increase the flexibility of a class by having a greater number of constructors for a single class. By have more than one way of initializing objects can be done using overloading constructors.

```
#include<iostream>
using namespace std;
class copy {
   int var, fact, real;
   public:
   copy(){}
   copy(int temp)  {
      var=temp;
   }
   copy(int a, int b){
      fact=a;
      real=b;
   }
};
int main() {
   int n,m;
   cout<<"Enter the Number : ";
   cin>>n>>m;
   copy obj;
   copy obj1(n);
   copy obj2(n, m);
   return 0;
}
```

Q14.Define Object as Function Argument with syntax example.

Answer---:

In C++ programming, objects can be passed to function in similar way as variables and structures. The syntax and procedure to return object is similar to that of returning structure from function.



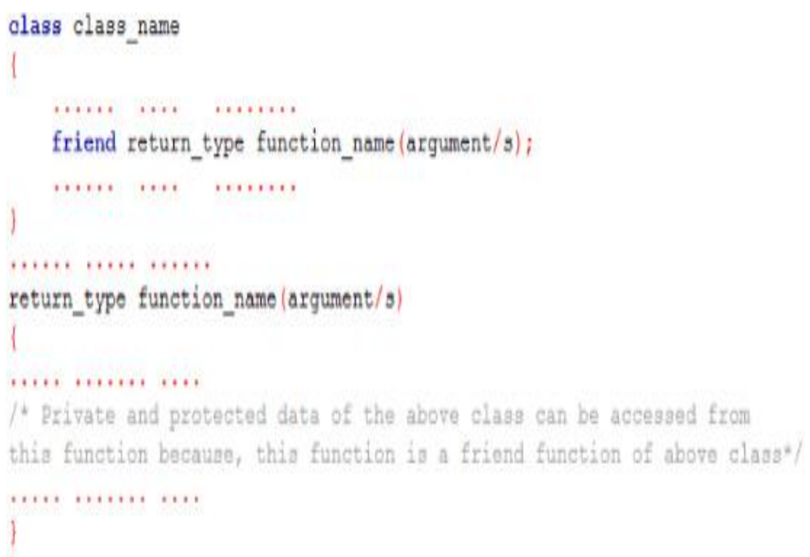
```
..... .... .....
class class_name
{
  .... ... ....
  return_type function_name(class_name para1, class_name para2)
  {
   ..... ... .....
  }
  ...... ..... .......
}

main()
{
 class_name obj1, obj2, obj3;

  obj1.function_name(obj2,obj3 )
  ..... ... ....
}
```

Figure: Passing Object to Function

Q15.Define Friend Function with basic syntax example. What are the characteristics of friend function?

Answer---:

A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.



```
class class_name
{
    ...... ....  ........
    friend return_type function_name(argument/s);
    ...... ....  ........
}
...... ..... ......
return_type function_name(argument/s)
{
..... ....... ....
/* Private and protected data of the above class can be accessed from
this function because, this function is a friend function of above class*/
..... ....... ....
}
```

Characteristics:

- A friend function is not in the scope of the class in which it has been declared as friend.
- It cannot be called using the object of that class.
- It can be invoked like a normal function without any object.
- Unlike member functions, it cannot use the member names directly.
- It can be declared in public or private part without affecting its meaning.
- Usually, it has objects as arguments.

Q16.Define Friend Class with basic syntax example.

Answer---:

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class. For example,

```
#include <iostream>
using namespace std;
class TwoValues {
int a;
int b;
public:
TwoValues(int i, int j) { a = i; b = j; }
friend class Min;
};
class Min {
public:
int min(TwoValues  x);
};
```

```
int Min::min(TwoValues  x)
{
return x.a < x.b ? x.a : x.b;
}
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
return 0;
}
```

Q17.Define Copy Constructor. Purposes of copy constructor. Give example with basic syntax of copy constructor.

Answer---:

Copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function .
- Copy an object to return it from a function.
- If a copy constructor is not defined in a class the compiler itself defines one (default copy constructor).
- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
- It must make a deep copy with an explicitly defined copy constructor.

Example:

```
class Example        {
    int a,b;
    public:
    Example(int x,int y) {
    a=x;
    b=y;
    cout<<"\nIm Constructor";
    }

    void Display()    {
    cout<<"\nValues :"<<a<<"\t"<<b;
    }
};
```

```
int main()              {
       Example Object(10,20);
       //Copy Constructor
       Example Object2=Object;

      // Constructor invoked.
       Object.Display();
       Object2.Display();

       return 0;
}
```

Q18.How private members of a class can be accessed from outside of the class?

Answer---:

In C++, private members of a class are designed to be inaccessible from outside the class. This encapsulation is a fundamental principle of object-oriented programming. Private members of a class are not accessible directly from outside the class. However, there are several methods to access them. Here:

1. **Friend Keyword:** You can declare another class or function as a friend of the current class. This allows the friend to access the private members directly. For example:

```
class MyClass {
private:
   int secretValue;
public:
   MyClass() : secretValue(42) {}
   friend void revealSecret(MyClass& obj);
};
void revealSecret(MyClass& obj) {
   cout << "The secret value is: " << obj.secretValue << std::endl;
}
```

2. **Public Getter Functions:** A common practice is to provide public member functions (getters) that return the values of private members. This maintains encapsulation while allowing controlled access.

```
class MyClass {
private:
   int secretValue;
public:
   MyClass() : secretValue(42) {}
   int getSecretValue() const {
      return secretValue;
   }
};
```

Q19.What is namespace? What is the purpose of namespace? Give an example of namespace.

Answer---:

A namespace in C++ is a declarative region that provides a scope to the identifiers (such as variables, functions, classes, etc.) inside it. Namespaces are used to organize code and prevent name conflicts, especially in large projects or when integrating multiple libraries.

Purpose of Namespace

- **Avoiding Name Conflicts:** Namespaces help avoid naming collisions by allowing the same name to be used in different contexts. For example, two different libraries can have a function named calculate(), but if they are in different namespaces, they can coexist without conflict.
- **Organizing Code:** Namespaces allow developers to group related code together, making it easier to manage and understand. This organization can improve code readability and maintainability.
- **Modularity:** By using namespaces, you can create modular code that can be reused across different projects without worrying about naming issues.

Example:

```
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}
namespace Science {
    int add(int a, int b) {
        return a + b + 1; // Different implementation
    }
}
int main() {
    int sum1 = Math::add(2, 3); // Calls Math's add
    int sum2 = Science::add(2, 3); // Calls Science's add
}
```