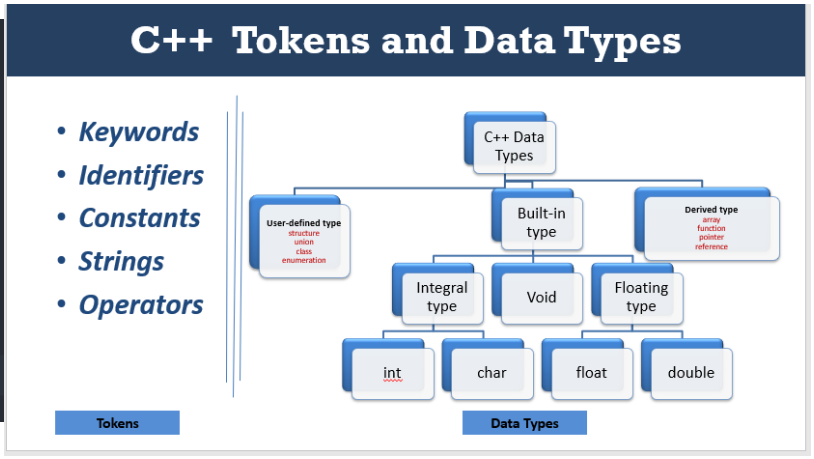
Q1. What is C++ Programming? Write a basic C++ code? Name some C++ Tokens and Data Types?

--- 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.

```
4   class Car {
5   
6   public:
7     string color; // Data member
8     string model; // Data member
9   };
```

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.

```
4   class Car {
5   public:
6     string color; // Data member
7     string model; // Data member
8   
9     void start() { // Member function
10      cout << "The car has started." << endl;
11    }
12  };
```

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;
  }
};
```

```
16  int main() {
17
18    Car car1;
19    car1.start();
20
21    return 0;
22  }
```

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
}
```

Q20. Difference between **Shallow Copy** & **Deep Copy**....

Aspect	Shallow Copy	Deep Copy
Definition	Creates a new object by copying references to the original object's data.	Creates a new object by copying the actual data of the original object.
Memory Allocation	Does not allocate new memory for the referenced objects; shares the same memory addresses.	Allocates new memory for all objects referenced, creating independent copies.
Data Sharing	Both the original and copied objects share the same data; changes in one affect the other.	The original and copied objects have their own copies of the data; changes in one do not affect the other.
Use Case	Suitable for lightweight objects where shared data is acceptable.	Necessary for complex objects where independent copies are required to avoid side effects.
Performance	Generally faster due to less memory allocation and copying.	Slower due to additional memory allocation and copying of data.
Implementation	Typically done using default copy constructors or assignment operators.	Requires custom copy constructors and assignment operators to ensure all data is copied.
Risk of Dangling Pointers	Higher risk if the original object is deleted, as the copied object may reference invalid memory.	Lower risk since each object manages its own memory independently.

Semester Final Questions

Q1. What is Inheritance? Why is inheritance used? / Describe features of inheritance. / Describe the importance of inheritance.

Answer---:

This mechanism of deriving a new class from existing/old class is called **inheritance**. The old class is known as “base” class, “super” class or “parent” class” and the new class is known as “sub” class, “derived” class, or “child” class.

Syntax:

```
class DerivedClass : accessSpecifier BaseClass {
   // Additional members of the derived class
};
```

Why Inheritance is Used

- **Code Reusability**: Inheritance allows new classes to reuse existing code, reducing redundancy and improving maintainability.
- **Hierarchical Classification**: It enables the creation of a class hierarchy, making it easier to organize and manage related classes.
- **Polymorphism**: Inheritance supports polymorphism, allowing objects of different classes to be treated as objects of a common superclass

Features of Inheritance

- **Subclassing**: New classes (subclasses) can inherit attributes and methods from existing classes (superclasses).
- **Relationship**: Establishes a relationship where a subclass is a specialized version of a superclass.
- **Method Overriding**: Subclasses can provide specific implementations of methods defined in their superclasses.

Importance of Inheritance

- **Simplifies Code Management**: By allowing shared functionality, inheritance simplifies code updates and maintenance.
- **Enhances Flexibility**: It provides a way to extend existing code without modifying it, promoting flexibility in software design.
- **Facilitates Collaboration**: Inheritance allows teams to work on different subclasses while maintaining a consistent interface.

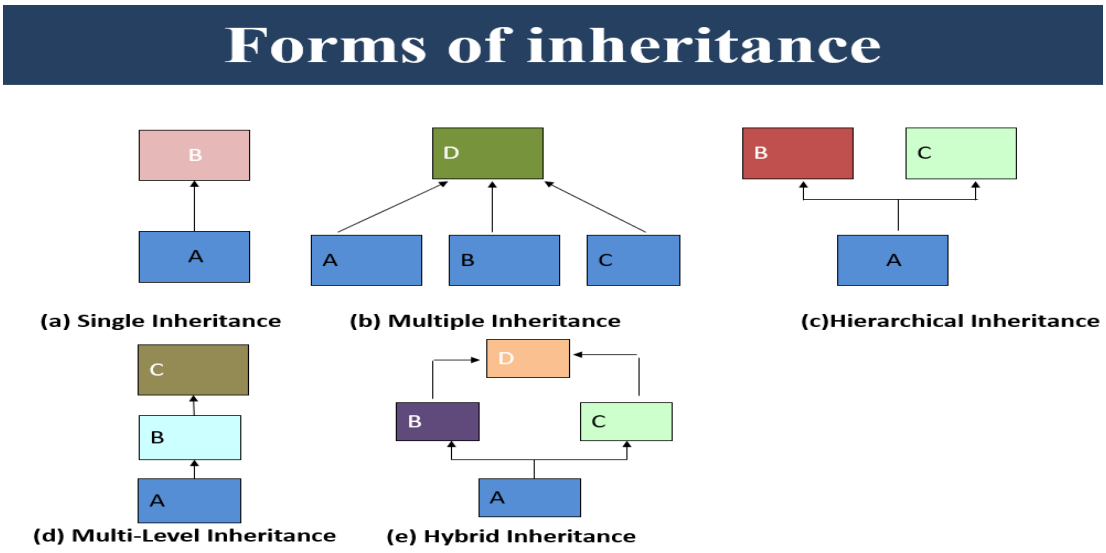
Q2. Describe different forms of inheritance.

Answer---:

Types of Inheritance:

- **Single Inheritance**: One class inherits from one base class.
- **Multiple Inheritance**: A class inherits from multiple base classes.
- **Multilevel Inheritance**: A derived class inherits from another derived class.
- **Hierarchical Inheritance**: Multiple classes inherit from the same base class.
- **Hybrid Inheritance**: A combination of single, hierarchical and multilevel inheritance.

(picture is only for better understanding.)



Extended---:

- **Single Inheritance:** When a single derived class is created from a single base class then the inheritance is called as single inheritance.

- **Multiple Inheritance**: When a derived class is created from more than one base class then that inheritance is called as multiple inheritance.

- **Multilevel Inheritance**: When a derived class is created from another derived class, then that inheritance is called as multilevel inheritance.

- **Hierarchical Inheritance**: When more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.

- **Hybrid Inheritance**: Any combination of single, hierarchical and multilevel inheritances is called as hybrid inheritance.

Q3. Explain how access specifiers (public, protected, private) impact inheritance in C++. Provide examples to show how changing an access specifier affects the accessibility of base class members in a derived class.

Answer---:

In C++, access specifiers (public, protected, and private) determine how members of a base class can be accessed in a derived class. This structure allows developers to control the visibility and accessibility of class members, which is crucial for encapsulation and maintaining a clean interface in object-oriented programming. Here's a brief overview:

- **Public Inheritance**: Members retain their access levels.

- **Protected Inheritance**: Members become protected, restricting access.
- **Private Inheritance**: All members become private, limiting access even further.

Access Specifier (data members)	Public Derivation	Private Derivation	Protected Derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Q4. Describe the "Diamond problem" in multiple inheritance and how C++ resolves this issue using virtual inheritance. Write a C++ program demonstrating the diamond problem with a base class Person, and derived classes Student and Employee, with TeachingAssistant inheriting from both Student and Employee.

Answer---:

The Diamond Problem occurs when a derived class inherits from two base classes, which both inherit from a common base class. This creates ambiguity because the derived class ends up with two copies of the common base class. C++ uses virtual inheritance to avoid duplicate copies of the base class. When a base class is declared as virtual, it ensures only a single shared instance of the base class is used, eliminating redundancy.

```
#include <iostream>

using namespace std;
```

```
class Person {
public:
   string name;
   void setName(string name) {
      this->name = name;
   }
   void display() const {
      cout << "Name: " << name << endl;
   }
};
class Student : virtual public Person {
public:
   void study() {
      cout << name << " is studying." << endl;
   }
};
class Employee : virtual public Person {
public:
   void work() {
      cout << name << " is working." << endl;
   }
};
class TeachingAssistant : public Student, public Employee {
public:
   void assist() {  cout << name << " is assisting in teaching." << endl;  }
};
```

```
int main() {
    TeachingAssistant ta;
    ta.setName("Abdullah Tamimn");
    ta.display();
    ta.study();
    ta.work();
    ta.assist();
    return 0;
}
```

Q5. Describe the order in which constructors and destructors are called in a C++ inheritance hierarchy. Given a base class Animal and derived classes Mammal and Dog (where Dog inherits from Mammal), explain the order of constructor and destructor calls when object of Dog is created and then destroyed with code.

Answer---:

In a C++ inheritance hierarchy, **constructors** and **destructors** are called in a specific order:

1. **Constructor Call Order**:
    - Constructors are called **from the base class to the derived class**.
    - First, the constructor of the **base class** is executed.
    - Then, the constructor of the intermediate derived class is executed (if any).
    - Finally, the constructor of the most derived class is executed.
2. **Destructor Call Order**:
    - Destructors are called **in reverse order**, starting with the most derived class and ending with the base class.

Given Classes: Animal (Base), Mammal (Intermediate), and Dog (Derived)

When an object of class Dog is created:

1. The **constructor of Animal** (base class) is called first.
2. Then, the **constructor of Mammal** (intermediate derived class) is called.
3. Finally, the **constructor of Dog** (most derived class) is called.

When the object is destroyed:

1. The **destructor of Dog** (most derived class) is called first.
2. Then, the **destructor of Mammal** is called.
3. Finally, the **destructor of Animal** is called.

```
#include <iostream>
using namespace std;
class Animal {
public:
   Animal() {
      cout << "Animal constructor called." << endl;
   }
   ~Animal() {
      cout << "Animal destructor called." << endl;
   }
};
class Mammal : public Animal {
public:
   Mammal() {
      cout << "Mammal constructor called." << endl;
   }
   ~Mammal() {
```

```
        cout << "Mammal destructor called." << endl;
    }
};
class Dog : public Mammal {
public:
    Dog() {
        cout << "Dog constructor called." << endl;
    }
    ~Dog() {
        cout << "Dog destructor called." << endl;
    }
};
int main() {
    Dog d;
    return 0;
}
```

Q6. What is operator overloading in C++? How many types of operators can be overloaded?

Answer---:

Operator overloading in C++ allows you to define or change the behavior of operators (+, -, *, = etc.) for user-defined types (e.g., classes). This enables operators to work intuitively with objects in the same way they work with fundamental types.

Syntax:

```
class ClassName {
public:
   ReturnType operator OperatorSymbol (Arguments) {
      // Define custom behavior here
   }
};
```

Two types of operators can be overloaded:

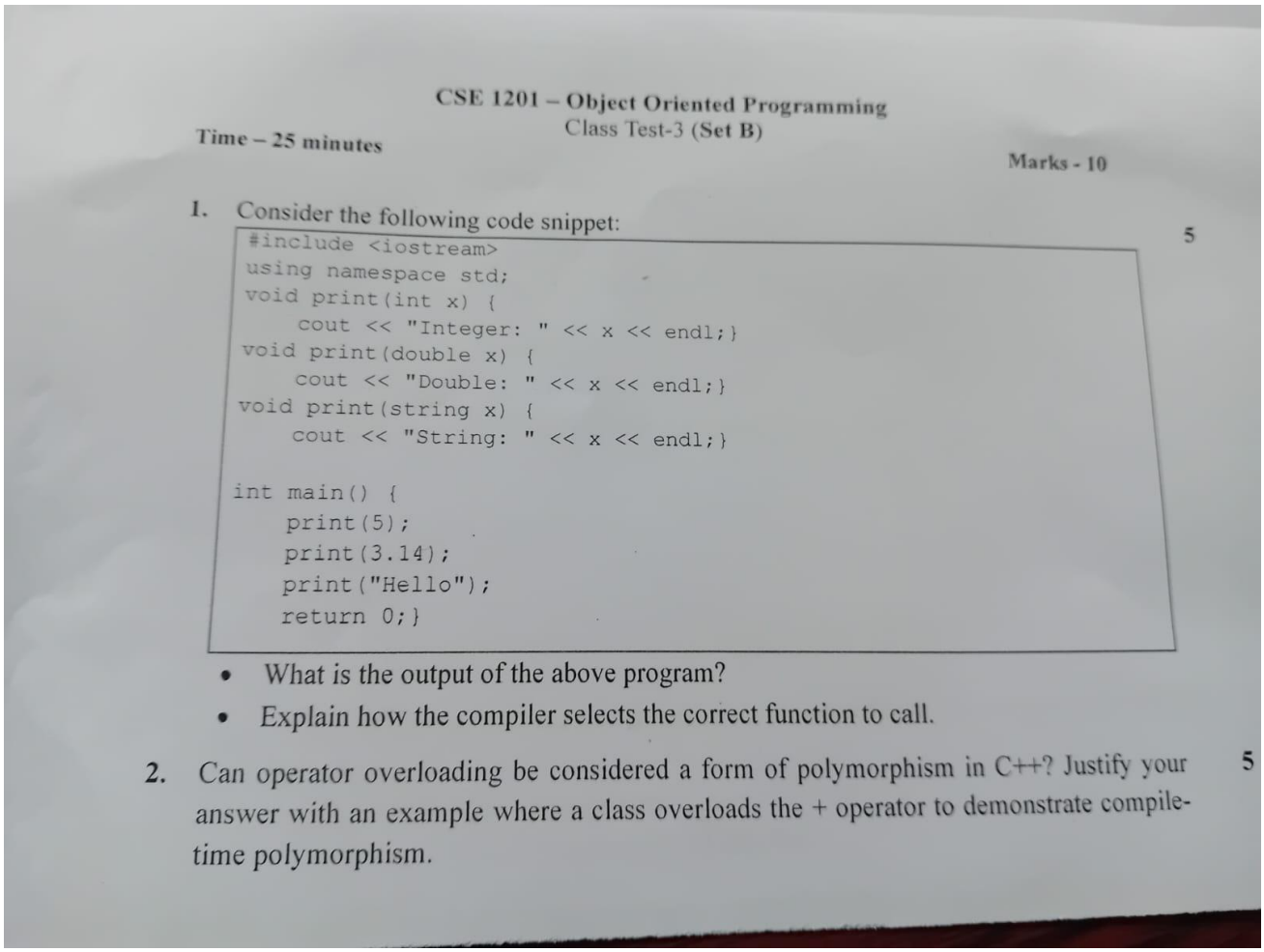
1. Unary Operator
2. Binary Operator

Q7. Why Operator overloading is used in C++ programming?

Answer---:

- **Enhances Code Readability**: Makes code more intuitive and easier to understand by allowing operators to work with user-defined types.
- **Improves Usability**: Allows objects of a class to behave like built-in types with operators (+, -, etc.).
- **Simplifies Complex Expressions**: Avoids verbose method calls (e.g., c1 + c2 instead of c1.add(c2)).
- **Custom Behavior**: Provides flexibility to define specific behavior for operators when applied to objects.
- **Encapsulation and Reusability**: Keeps operator-related functionality inside the class, maintaining encapsulation.
- **Supports Object-Oriented Design**: Helps make user-defined types more natural and intuitive in object-oriented programming.

Q8. Solve these questions.



CSE 1201 – Object Oriented Programming
Class Test-3 (Set B)

Time – 25 minutes

Marks - 10

1. Consider the following code snippet: 5

```
#include <iostream>
using namespace std;
void print(int x) {
    cout << "Integer: " << x << endl;}
void print(double x) {
    cout << "Double: " << x << endl;}
void print(string x) {
    cout << "String: " << x << endl;}

int main() {
    print(5);
    print(3.14);
    print("Hello");
    return 0;}
```

- What is the output of the above program?
- Explain how the compiler selects the correct function to call.

2. Can operator overloading be considered a form of polymorphism in C++? Justify your answer with an example where a class overloads the + operator to demonstrate compile-time polymorphism. 5

Answer---1:

Integer: 5

Double: 3.14

String: Hello

The compiler determines which print function to call based on the **type of the argument** passed during the function call:

print(5) matches void print(int x) because 5 is an integer.

print(3.14) matches void print(double x) because 3.14 is a double.

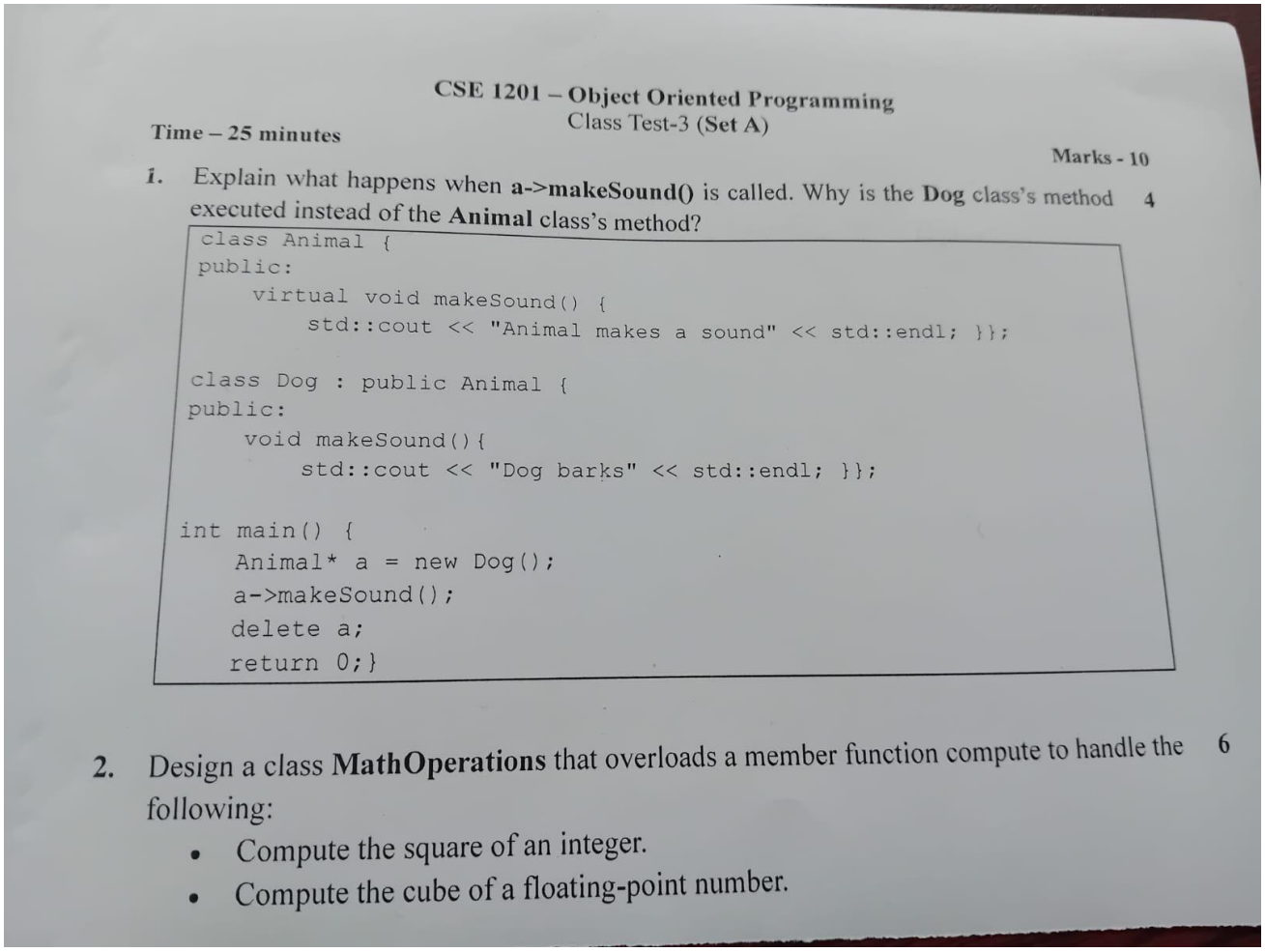
print("Hello") matches void print(string x) because "Hello" is a string literal, which is implicitly converted to a string.

Answer---2:

Operator Overloading is a form of compile-time polymorphism in C++ because it allows the same operator to behave differently based on the types of operands. By overloading operators, we can define custom behavior for operators like +, -, *, etc., for user-defined types (e.g., classes).

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

class Complex {
private:
   double real, imag;

public:
   Complex(double r = 0, double i = 0) : real(r), imag(i) {}

   Complex operator+(const Complex &c) {
       Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
   }

   void display() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(3.0, 4.0), c2(1.0, 2.0);
   Complex c3 = c1 + c2;
   c3.display();
   return 0;
}
```

Q9. Solve these questions.



CSE 1201 – Object Oriented Programming
Class Test-3 (Set A)

Time – 25 minutes

Marks - 10

1. Explain what happens when **a->makeSound()** is called. Why is the **Dog** class's method executed instead of the **Animal** class's method? 4

```
class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal makes a sound" << std::endl; }};

class Dog : public Animal {
public:
    void makeSound(){
        std::cout << "Dog barks" << std::endl; }};

int main() {
    Animal* a = new Dog();
    a->makeSound();
    delete a;
    return 0;}
```

2. Design a class **MathOperations** that overloads a member function compute to handle the following: 6
   - Compute the square of an integer.
   - Compute the cube of a floating-point number.

Answer---1:

When a->makeSound() is executed, the Dog class's makeSound() method is executed instead of the Animal class's makeSound() method. This happens because the makeSound() function in the Animal class is declared as virtual.

In C++, a virtual function allows for runtime polymorphism, which ensures that the derived class's version of the function is called when accessed through a pointer or reference to the base class. In this case, calling

a->makeSound() dynamically binds the function call to the makeSound() implementation of the Dog class, because a points to an object of type Dog.

The line Animal* a = new Dog(); creates a pointer a of type Animal, but it points to an object of type Dog. When a->makeSound() is called, the virtual function mechanism ensures that the makeSound() function of the Dog class is executed, not the base class function.

Answer---2:

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

class MathOperations {
public:
   int compute(int x) {
      return x * x;
   }
   double compute(double x) {
      return x * x * x;
   }
};

int main() {
   MathOperations math;

   cout << "Square of 5: " << math.compute(5) << endl;
   cout << "Cube of 2.5: " << math.compute(2.5) << endl;

   return 0;
}
```

The compute() function is overloaded with two implementations. One takes an integer as input and computes its square. The other takes a floating-point number (double) and computes its cube. The compiler determines which version of compute() to call based on the type of the argument passed.

Q10. What is polymorphism? Why polymorphism is used? / Describe the features of polymorphism.

Answer---:

Polymorphism means one name having multiple forms. Polymorphism is the ability to create a variable, a function or an object that has more than one form.

The main 5 benefits of polymorphism are:

1. **Code Reusability:** Polymorphism allows for the reuse of code across different classes, reducing redundancy and improving maintainability.

2. **Flexibility:** Polymorphism enables objects of different types to be treated as objects of a common superclass, allowing for more flexible and dynamic code.

3. **Ease of Maintenance:** Changes in the base class can propagate to derived classes, making it easier to maintain and update code.

4. **Improved Readability:** Code that utilizes polymorphism can be more intuitive and easier to understand, as it allows for a common interface for different implementations.

5. **Dynamic Binding:** Polymorphism supports dynamic method resolution, meaning the method that gets executed is determined at runtime, allowing for more adaptable code.

Q11. What are the types of polymorphism?

Answer---:

There are two types of polymorphism. They are:

Compile-time polymorphism:

- In compile time polymorphism, compiler is able to select the appropriate function a particular call at the compile time.

Run-time polymorphism:

- In run time polymorphism, an appropriate member function is selected while the program is running.

Q12. Describe about Function overriding.

Answer---:

Function overriding occurs when a **derived class** provides a specific implementation for a function that is already defined in its **base class**. It allows the derived class to modify or extend the behavior of a base class function.

Key Characteristics of Function Overriding

1. **Inheritance**:
    - Function overriding can only occur in a class hierarchy where a derived class inherits from a base class.
2. **Same Function Signature**:
    - The overridden function in the derived class must have the same name, return type, and parameters as the function in the base class.
3. **Virtual Functions**:
    - The base class function must be marked as virtual to enable function overriding.

- This ensures that the function called is based on the object's actual type (runtime polymorphism).

4. **Runtime Polymorphism**:
   - Function overriding supports **runtime polymorphism**, allowing behavior to be determined at runtime.

Q13. Differences Between Overloading and Overriding?

Answer---:

Differences Between Overloading and Overriding:

Feature	Function Overloading	Function Overriding
Purpose	Same function name, different signatures (compile-time).	Modify behavior of base class function (runtime).
Polymorphism	Compile-time polymorphism.	Runtime polymorphism.
Inheritance	Not required.	Required.

Q14. Describe about Virtual function and Pure Virtual Function.

Answer---:

Virtual Function:

Virtual function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function.

- Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
- Virtual Keyword is used to make a member function of the base class Virtual

Syntax is :

```
virtual double areaCalculation()  {  return 0;}
```

Pure Virtual Function:

The compiler that the function has no body and above virtual function will be called **pure virtual function**.

Syntax:

```
virtual double area_calc()=0;
```

```
class Shape//Base class
{
public:
Shape() {}
 virtual double area_calc() =0;
};
```

Q15. What is static and dynamic binding? What are the differences between static and dynamic polymorphism?

Answer---:

Early Binding:

- Connecting the function call to the function body is called Binding. When it is done before the program is run, its called Early Binding or Static Binding or Compile-time Binding.

Late Binding:

- In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called Dynamic Binding or Runtime Binding.

Differences Between Static and Dynamic Polymorphism

Aspect	Static Polymorphism	Dynamic Polymorphism
Binding Time	Compile time	Runtime
Implementation	Achieved through **function overloading** and **operator overloading**.	Achieved through **function overriding** and **virtual functions**.
Performance	Faster, as function calls are resolved at compile time.	Slower, as function calls are resolved at runtime.
Flexibility	Less flexible, requires different function signatures.	More flexible, allows overriding of base class functions.
Keyword Usage	Does not require any special keywords.	Requires the `virtual` keyword in the base class.
Example	**Function Overloading**: Multiple versions of a function with different parameters.	**Function Overriding**: Derived class overrides a virtual function from the base class.

Q16. What is Abstract class? Describe the characteristics of abstract class.

Answer---:

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

The main characteristics of an Abstract Class in C++ are:

1. **Cannot be Instantiated**: Abstract classes cannot be used to create objects. They are meant to be inherited by other classes.

2. **May Contain Abstract Methods**: Abstract classes can have abstract methods, which are declared without any implementation. Derived classes must provide the implementation for these abstract methods.

3. **Inheritance and Overriding**: Abstract classes can have both abstract and non-abstract (concrete) methods. Derived classes must override the abstract methods and provide their own implementation.

4. **Access Modifiers**: Abstract classes can have access modifiers (public, protected, private) for their members, just like regular classes.

5. **Partial Implementation**: Abstract classes can provide partial implementation of the functionality, leaving some methods abstract for the derived classes to implement.

Q17. What is Exception Handling? Explain the roles of try, catch and throw keywords with examples.

Answer---:

An exception is a problem that arises during the execution of a program. **Exception handling in C++** is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero

C++ exception handling is built upon three keywords: try, catch, and throw.

- **try**: Contains the code that may throw an exception.
- **throw**: Used to raise an exception when an error occurs.
- **catch**: Catches and handles the exception thrown by the throw.

Example:

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       system("cls");
6
7       try {
8           int a = 10, b = 0;
9           if (b == 0) {
10              throw 0;
11          }
12          cout << a / b << endl;
13      }
14      catch (int e) {
15          cout << "Division by " << e << " is not allowed " << endl
16      }
17      return 0;
18  }
19
```

Q18. Define Static Data Member and Static Member function with an example.

Answer---:

Static Data Member:

A static data member is a variable that is shared among all objects of a class. It is declared using the static keyword inside the class and must be **initialized outside the class**. It belongs to the class, not to any specific object, and is stored in a separate memory location.

Static Member Function:

A static member function is a function that can be called without creating an object of the class. It can access only **static data members** of the class. Declared using the static keyword.

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

class MyClass {
private:
   static int count; // Static data member
public:
   MyClass() { count++; } // Constructor increments count

   static void showCount() { // Static member function
      cout << "Count: " << count << endl;
   }
};

// Initialize the static data member
int MyClass::count = 0;
```

```
int main() {
   MyClass obj1, obj2; // Two objects created
   MyClass::showCount(); // Access static function without an object

   return 0;
}
```

Q19. Define Static Data Member and Static Member function with an example.

Answer---:

Dynamic memory allocation allows programs to allocate memory at **runtime**, instead of compile time. It provides flexibility for managing memory efficiently based on program requirements.

Uses of new and delete Keywords:

1. **new Keyword**:
   - Allocates memory dynamically on the **heap**.
   - Returns a pointer to the allocated memory.
   - Can be used to allocate memory for single variables or arrays.
2. **delete Keyword**:
   - Frees memory allocated with new.
   - Prevents memory leaks by deallocating unused memory.

For example:

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

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

Q20. What is Upcasting and Downcasting?

Answer---:

Upcasting:

It is the process to create the derived class's pointer or reference from the base class's pointer or reference, and the process is called Upcasting. It means the upcasting used to convert the reference or pointer of the derived class to a base class.

```
Base *ptr = &derived_obj;
```

Downcasting:

The Downcasting is an opposite process to the upcasting, which converts the base class's pointer or reference to the derived class's pointer or reference. It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast.

```
Derived *d_ptr = &b_obj;
```

Q21. What is Dynamic Constructor?

Answer---:

Dynamic Constructor:

When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as dynamic constructor. By using this, we can dynamically initialize the objects.

Q22. What is Virtual Destructor? Why we use virtual destructor?

Answer---:

A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object.

When a pointer object of the base class is deleted that points to the derived class, only the parent class destructor is called due to the early bind by the compiler. In this way, it skips calling the derived class' destructor, which leads to memory leaks issue in the program. A base or parent class destructor use the **virtual** keyword preceded by the destructor tilde (~) sign that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Q22. What is Template in C++? Define types of templates with syntax example.

Answer---:

Templates in C++ are a feature that allows the creation of **generic functions** or **classes** that can work with any data type. They enable code reusability and type independence, avoiding the need to write duplicate code for different data types.

Templates can be represented in two ways:

- Function templates
- Class templates

Function Templates:

A **function template** allows to create a single function that can operate on different data types.

Syntax: template <class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}

Class Template:

A **class template** allows to define a blueprint for classes that can handle any data type.

Syntax: template<class Ttype>
class class_name
{
   // body of class.
}