Chapter 1:

1. What are the merits and demerits of object-oriented programming? Briefly describe the features of C++. (4+4)

Answer:

Merits:

- Modularity and code reusability.
- Easier maintenance and scalability.
- Enhanced data security with encapsulation.
- Closer representation of real-world entities.

Demerits:

- Increased program complexity.
- Higher memory and processing requirements.
- Steeper learning curve.

Features of C++:

- Supports OOP concepts like classes, inheritance, and polymorphism.
- Includes both procedural and object-oriented programming.
- Offers operator overloading and dynamic memory management.

2. What is object-oriented programming? Differentiate it from procedure-oriented programming. (2+6)

Answer:

Object-Oriented Programming (OOP): A paradigm where real-world entities are represented as objects, focusing on data encapsulation, abstraction, inheritance, and polymorphism.

Differences:

- OOP: Emphasizes objects and data.
- POP: Focuses on functions and logic.
- OOP: Encapsulation hides data; data is secure.
- POP: Global data is less secure.
- OOP: Suitable for large, scalable systems.
- POP: Best for small, straightforward applications.

3. Explain encapsulation and its importance in object-oriented programming. (4 marks)

Answer:

Encapsulation is the bundling of data and methods into a class and restricting direct access using access specifiers (private, protected, public).

Importance:

- Protects data integrity by controlling access.
- Simplifies debugging and maintenance.
- Promotes modularity and better code organization.

4. Why is OOP considered better than procedural programming for large projects? (6 marks)

Answer:

- Promotes modularity: Divides complex projects into classes.
- Ensures code reusability: Inheritance allows use of existing code.
- Improves data security: Encapsulation restricts data access.
- Enhances scalability: Supports system expansion without major disruptions.
- Represents real-world problems more effectively, simplifying design and understanding.

3. What do you mean by enumerators and reference variables?

Enumerators: Enumeration is a user-defined data type consisting of named integral constants. Example:

cpp

Copy code

enum Color { RED, GREEN, BLUE };

Reference Variables: A reference variable acts as an alias for an existing variable, providing another name to access the same memory location. Example:

cpp

Copy code

int x = 10;

int &ref = x;

ref = 20; // Updates x to 20

4. Define tokens and preprocessor directives. What do you mean by inline function? Why do we need inline functions?

Tokens: Tokens are the smallest elements of a program, such as keywords, identifiers, literals, operators, and punctuators.

Preprocessor Directives: Instructions starting with # executed before the compilation (e.g., #include, #define).

Inline Function: A function expanded at compile time using the inline keyword.

Need for Inline Functions: They eliminate function call overhead, making small functions faster by embedding their code directly into the calling code.

5. What is a constructor? Write some characteristics of the constructor function.

Constructor: A special member function that initializes objects automatically when they are created. It shares the class name.

Characteristics:

1. Invoked automatically during object creation.
2. No return type, not even void.
3. Can be overloaded.
4. Default constructors are provided if no constructor is defined.

6. What do you mean by object and class? For what purpose do you use static data?

Class: A blueprint or template for creating objects.

Object: An instance of a class that contains attributes and methods.

Static Data: Shared across all objects of the class and retains its value between function calls. It is useful for counting objects or maintaining shared information.

7. Write some rules for virtual functions.

Rules for Virtual Functions:

1. Must be declared in the base class using the virtual keyword.
2. Can be overridden in the derived class.
3. Cannot be static or a friend function.
4. A class with virtual functions should have a virtual destructor.
5. Accessed through pointers or references to the base class.

8. Define operator overloading and list out the not-overloadable operators in C++.

Operator Overloading: Extending the functionality of operators to work with user-defined types.

Not-Overloadable Operators:

1. Scope resolution (::)
2. Member access (.)
3. Pointer-to-member (.*)
4. Conditional (?:)
5. Sizeof operator

9. Briefly explain the file stream class hierarchy. Define file pointer and write about its types.

File Stream Class Hierarchy:

1. ios: Base class for all input/output operations.
2. istream: For input stream (e.g., cin).
3. ostream: For output stream (e.g., cout).
4. ifstream: For file input.
5. ofstream: For file output.
6. fstream: For both file input and output.

File Pointer: Points to the current position in the file.

Types:

1. get pointer (gptr): Used in input operations.
2. put pointer (pptr): Used in output operations.

10. How are the constructors and destructors invoked in multiple inheritances?

Answer: In multiple inheritances, constructors of the base classes are invoked in the order of their declaration in the derived class. The derived class constructor is executed after the base class constructors. Destructors are called in the reverse order—derived class first, followed by base classes.

11. How do templates exhibit generic programming?

Answer: Templates allow writing generic and reusable code that works with any data type, eliminating redundancy.

Syntax:

- Function Template:
- cpp
- Copy code
- template <typename T>
- T add(T a, T b) { return a + b; }
- Class Template:
- cpp
- Copy code
- template <typename T>
- class MyClass {
- T data;
- };

12. Suppose the base class pointer is also a pointer to the derived class object, and both classes have constructors and destructors. Then what happens if the object pointer is deleted?

Answer: If the base class pointer points to a derived class object and the object is deleted without a virtual destructor in the base class, only the base class destructor is invoked. This can lead to resource leakage. Using a virtual destructor ensures both destructors (base and derived) are called properly.

Detailed Answers to Theory Questions

1. A) What is the significance of namespace in C++? Define dynamic memory allocation and write its syntax.

Significance of Namespace in C++:

Namespaces prevent name conflicts in large programs or when using multiple libraries by providing a way to group logically related classes, objects, and functions under a unique name. The std namespace is commonly used in C++ for standard library functions.

Dynamic Memory Allocation (DMA):

Dynamic Memory Allocation allows allocating memory at runtime, giving flexibility to create variables or arrays of the desired size when the program executes.

Syntax:

Allocate memory: type* ptr = new type; or type* ptr = new type[size];

Deallocate memory: delete ptr; or delete[] ptr;

1. B) Write the rules for operator overloading.

Overloading must not change the operator's precedence or associativity.

Operators like ::, ., sizeof, .*, and ?: cannot be overloaded.

At least one operand must be a user-defined type.

Cannot create new operators.

Overloaded operators can be member functions or non-member (friend) functions.

Assignment (=), function call (()), subscript ([]), and pointer dereference (->) operators must be member functions.

2. A) Differentiate between text files and binary files. How do you test errors in file operations?

Differences between Text Files and Binary Files:

Text Files: Store data in a human-readable format; typically use ASCII or Unicode encoding.

Binary Files: Store data in raw, binary format and are more compact but not human-readable.

Testing Errors in File Operations:

Use file stream functions like is_open() to check if the file opened successfully, fail() to test for general file operation errors, and eof() to check if the end of a file is reached.

3. A) What is inheritance? Why is it necessary? How do you remove ambiguity errors in multiple inheritance?

Inheritance: A mechanism for deriving new classes (child) from existing ones (parent), enabling code reuse and adding functionality.

Necessity: It reduces redundancy by allowing shared attributes and methods between classes, making the code easier to maintain and extend.

Removing Ambiguity Errors in Multiple Inheritance: Use virtual inheritance or explicitly specify which base class's member to call using the scope resolution operator.

3. B) Define virtual functions and abstract classes and write their syntax.

Virtual Functions: Member functions in a base class that can be overridden in derived classes using the virtual keyword.

Syntax:

class Base {

virtual void display() { cout << "Base"; }

};

Abstract Classes: Classes that cannot be instantiated and contain at least one pure virtual function.

Syntax:

class AbstractClass {

3. B) Define virtual functions and abstract classes and write their syntax.

Virtual Functions: Member functions in a base class that can be overridden in derived classes using the virtual keyword.

Syntax:

```
class Base {
    virtual void display() { cout << "Base"; }
};
```

Abstract Classes: Classes that cannot be instantiated and contain at least one pure virtual function.

Syntax:

```
class AbstractClass {
    virtual void func() = 0; // Pure virtual function
};
```

4. A) Define constructors and destructors. Write the types of constructors and explain them with their syntax.

Constructor: A special member function automatically called when an object is created.

Destructor: A special member function automatically called to free resources when an object goes out of scope.

Types of Constructors:

Default Constructor: No arguments.

```
class MyClass {
    MyClass() { cout << "Default"; }
};
```

Parameterized Constructor: Takes arguments.

```
class MyClass {
    MyClass(int x) { cout << x; }
};
```

Copy Constructor: Creates a copy of an object.

```
class MyClass {
    MyClass(const MyClass &obj) { ... }
};
```

4. B) What is a template? Why are templates necessary?

Template: A feature in C++ that allows functions and classes to operate with generic types, enabling code reuse.

Syntax:

```
template <typename T>
T add(T a, T b) { return a + b; }
```

Necessity: Templates provide a way to write generic and reusable code, minimizing redundancy and enabling functions or classes to work with any data type.

What is Inheritance?

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class, called the derived class or child class, to acquire the properties (data members) and behaviors (member functions) of an existing class, called the base class or parent class. It facilitates code reuse, promotes modularity, and supports the extension of existing functionality without modifying the original class.

Types of Inheritance in C++ with Syntax

Single Inheritance

A derived class inherits from a single base class.

Syntax:

```
class Base {
    // Base class members
};
class Derived : public Base {
    // Derived class members
}
```

Multiple Inheritance

A derived class inherits from two or more base classes.

Syntax:

```
class Base1 {
    // Members of Base1
};
class Base2 {
    // Members of Base2};
class Derived : public Base1, public Base2 {
    // Derived class members};
```

Multilevel Inheritance

A class is derived from another derived class, creating a chain of inheritance.

Syntax:

```
class Base {
    // Base class members};
class Intermediate : public Base {
    // Members of Intermediate class};
class Derived : public Intermediate {
    // Derived class members
};
```

Hierarchical Inheritance

Multiple derived classes inherit from a single base class.

Syntax:

```
class Base {
    // Base class members
};
class Derived1 : public Base {
    // Derived1 class members
;
class Derived2 : public Base {
    // Derived2 class members
};
```

Hybrid Inheritance

A combination of two or more types of inheritance, such as hierarchical and multiple inheritance.

Syntax:

```
class Base {
    // Base class members};
class Derived1 : public Base {
    // Derived1 class members};
class Derived2 {
    // Members of Derived2};
class HybridDerived : public Derived1, public Derived2 {
};
```

1. Write a program to create a class Student with attributes name, age, and grade. Add a member function to display the student details.

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

class Student {
    string name; int age, grade;
public:
    void setData(string n, int a, int g) { name = n; age = a; grade = g; }
    void display() { cout << "Name: " << name << ", Age: " << age << ", Grade: " << grade << endl; }
};

int main() {
    Student s;
    s.setData("John", 20, 90);
    s.display();
    return 0;
}
```

2. Write a program to create a class Rectangle with length and breadth. Add member functions to calculate area and perimeter.

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

class Rectangle {
    int length, breadth;
public:
    Rectangle(int l, int b) : length(l), breadth(b) {}
    void display() {
        cout << "Area: " << length * breadth << ", Perimeter: " << 2 * (length + breadth) << endl;
    }
};

int main() {
    Rectangle r(5, 3);
    r.display();
    return 0;
}
```

3. Write a program to demonstrate the concept of static data members and static member functions using a class.

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

class Counter {
    static int count;
public:
    Counter() { count++; }
    static void display() { cout << "Count: " << count << endl; }
};
int Counter::count = 0;   int main() {
    Counter c1, c2, c3;
    Counter::display();
    return 0;
```

7. Write a program to create a base class Shape and derive two classes Circle and Rectangle. Implement the area() function in derived classes.

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

class Shape {
public:
    virtual double area() = 0; // Pure virtual function
};
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override { return M_PI * radius * radius; }
};
class Rectangle : public Shape {
    double length, width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() override { return length * width; }
};
int main() {
    Shape *c = new Circle(5), *r = new Rectangle(4, 6);
    cout << "Circle Area: " << c->area() << "\nRectangle Area: " << r->area() << endl;
    delete c; delete r;
    return 0;
}
```

8. Write a program to demonstrate multiple inheritance using two base classes and one derived class.

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

class ClassA {
public:
    void methodA() { cout << "Method from ClassA\n"; }
};
class ClassB {
public:
    void methodB() { cout << "Method from ClassB\n"; }
};
class Derived : public ClassA, public ClassB {
public:
    void methodC() { cout << "Method from Derived\n"; }
};
int main() {
    Derived obj;
    obj.methodA();
    obj.methodB();
    obj.methodC();
    return 0;
}
```

9. Write a program to demonstrate the concept of hierarchical inheritance using appropriate classes.

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

class Parent {
public:
    void commonMethod() { cout << "Method in Parent\n"; }
};

class Child1 : public Parent {
public:
    void specificMethod1() { cout << "Method in Child1\n"; }
};

class Child2 : public Parent {
public:
    void specificMethod2() { cout << "Method in Child2\n"; }
};

int main() {
    Child1 c1;
    Child2 c2;
    c1.commonMethod();
    c1.specificMethod1();
    c2.commonMethod();
    c2.specificMethod2();
    return 0;
}
```

4. Write a program to overload the + operator to add two complex numbers using a class.

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

class Complex {
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    Complex operator+(const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }
    void display() { cout << real << " + " << imag << "i\n"; }
};

int main() {
    Complex c1(3, 4), c2(1, 2), c3;
    c3 = c1 + c2;
    c3.display();
    return 0;
}
```

5. Write a program to overload the unary ++ operator (prefix) using a friend function in C++.

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

class Counter {
    int value;
public:
    Counter(int v = 0) : value(v) {}
    friend Counter& operator++(Counter& c); // Friend function declaration
    void display() { cout << "Value: " << value << endl; }
};

Counter& operator++(Counter& c) {
    ++c.value; // Increment the value
    return c;
}

int main() {
    Counter c(5);
    ++c;
    c.display();
    return 0;
}
```

6. Write a program to overload the binary - operator to subtract two objects of a class.

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

class Point {
    int x, y;
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {}
    Point operator-(const Point& p) {
        return Point(x - p.x, y - p.y);
    }
    void display() { cout << "(" << x << ", " << y << ")\n"; }
};

int main() {
    Point p1(5, 7), p2(3, 4), p3;
    p3 = p1 - p2;
    p3.display();
    return 0;
}
```

10. Write a program to demonstrate runtime polymorphism using virtual functions in a base and derived class.

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

class Base {
public:
    virtual void display() { cout << "Display from Base class\n"; }
};

class Derived : public Base {
public:
    void display() override { cout << "Display from Derived class\n"; }
};

int main() {
    Base* bptr;
    Derived d;
    bptr = &d;
    bptr->display(); // Calls Derived's display due to virtual function
    return 0;
}
```

11. Write a program to make an array of pointers to the base class and call the virtual functions using derived class objects.

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

class Base {
public:
    virtual void display() { cout << "Base display\n"; }
};

class Derived1 : public Base {
public:
    void display() override { cout << "Derived1 display\n"; }
};

class Derived2 : public Base {
public:
    void display() override { cout << "Derived2 display\n"; }
};

int main() {
    Base* arr[2];
    Derived1 d1;
    Derived2 d2;

    arr[0] = &d1;
    arr[1] = &d2;

    for (int i = 0; i < 2; i++)
        arr[i]->display(); // Calls appropriate display() based on object type
    return 0;
}
```

12. Write a program to demonstrate the concept of pure virtual functions and abstract classes.

```
#include <iostream>
using namespace std;
class AbstractBase {
public:
    virtual void display() = 0; // Pure virtual function
    virtual ~AbstractBase() {} // Virtual destructor
};

class Derived : public AbstractBase {
public:
    void display() override { cout << "Display from Derived class\n"; }
};

int main() {
    AbstractBase* ptr = new Derived();
    ptr->display(); // Calls Derived's display
    delete ptr; // Cleans up memory
    return 0;
}
```

13. Write a program to implement function templates for finding the maximum of two numbers of different data types.

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

template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max(3, 7): " << findMax(3, 7) << endl;
    cout << "Max(4.5, 2.8): " << findMax(4.5, 2.8) << endl;
    cout << "Max('A', 'B'): " << findMax('A', 'B') << endl;
    return 0;
}
```

14. Write a program to create a binary file to store and read employee records using classes.

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

template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max(3, 7): " << findMax(3, 7) << endl;
    cout << "Max(4.5, 2.8): " << findMax(4.5, 2.8) << endl;
    cout << "Max('A', 'B'): " << findMax('A', 'B') << endl;
    return 0;
}
```

15. Write a program to handle division by zero exceptions using exception handling in C++.

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

int main() {
    int a, b;
    cout << "Enter two numbers (numerator and denominator): ";
    cin >> a >> b;

    try {
        if (b == 0) throw "Division by zero error!";
        cout << "Result: " << (a / b) << endl;
    } catch (const char* e) {
        cout << "Exception: " << e << endl;
    }

    return 0;
}
```

16. Write a program to dynamically allocate memory for an array and find its largest element using pointers.

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

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    int* arr = new int[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) max = arr[i];
    }

    cout << "Largest element: " << max << endl;
    delete[] arr; // Free memory
    return 0;
}
```

1. Program to Demonstrate Constructor Overloading

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

class Rectangle {
    double length, width;
public:
    Rectangle() : length(1), width(1) {} // Default constructor
    Rectangle(double l, double w) : length(l), width(w) {} // Parameterized constructor
    double area() { return length * width; }
};

int main() {
  Rectangle r1; // Default constructor
  Rectangle r2(4.5, 3.2); // Parameterized constructor

  cout << "Area of default rectangle: " << r1.area() << endl;
  cout << "Area of parameterized rectangle: " << r2.area() << endl;
  return 0;
}
```

Program to Implement a Simple Class for Banking System

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

class BankAccount {
    string name;
    int accountNumber;
    double balance;
public:
    void initialize(string n, int acc, double bal) {
        name = n;
        accountNumber = acc;
        balance = bal;
    }
    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << ", New Balance: " << balance << endl;
    }
    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Insufficient balance!\n";
        } else {
            balance -= amount;
            cout << "Withdrawn: " << amount << ", Remaining Balance: " << balance << endl;
        }
    }
    void display() {
        cout << "Name: " << name << ", Account Number: " << accountNumber << ", Balance: " << balance <<
endl;
    }
};

int main() {
    BankAccount acc;
    acc.initialize("John Doe", 12345, 1000.0);
    acc.display();
    acc.deposit(500);
    acc.withdraw(200);
    acc.withdraw(1500);
    return 0;
}
```

3. Program to Implement a Matrix Multiplication

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

int main() {
    int a[10][10], b[10][10], c[10][10], r1, c1, r2, c2;

    cout << "Enter rows and columns of matrix A: ";
    cin >> r1 >> c1;
    cout << "Enter rows and columns of matrix B: ";
    cin >> r2 >> c2;

    if (c1 != r2) {
        cout << "Matrix multiplication not possible.\n";
        return 0;
    }

    cout << "Enter elements of matrix A:\n";
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c1; j++)
            cin >> a[i][j];

    cout << "Enter elements of matrix B:\n";
    for (int i = 0; i < r2; i++)
        for (int j = 0; j < c2; j++)
            cin >> b[i][j];

    // Multiply matrices
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            c[i][j] = 0;
            for (int k = 0; k < c1; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }

    cout << "Resultant matrix:\n";
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++)
            cout << c[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

4. Program to Swap Two Numbers Using Call by Reference

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

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;

    cout << "Before swapping: x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "After swapping: x = " << x << ", y = " << y << endl;

    return 0;
}
```