INSTITUTE OF REMOTE SENSING
ANNA UNIVERSITY, CHENNAI - 600 025.

LAB MANUAL
OBJECT ORIENTED PROGRAMMING LABORATORY

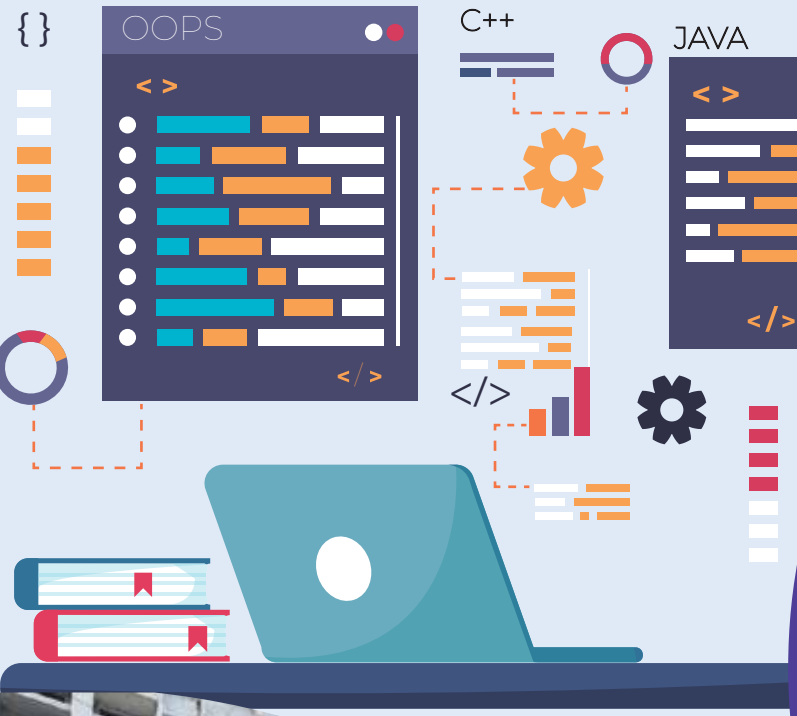

TABLE OF CONTENTS

Expt No. 1	ARITHMETIC OPERATIONS	Date of Expt:

AIM:

Perform basic arithmetic operations (addition, subtraction, multiplication, division, modulus) in C++.

SOFTWARE USED:

CodeBlocks

PROCEDURE:

1. Use basic arithmetic operators.
2. Perform operations on integers or floats.
3. Output the results for each operation.

Code 1: Addition of two numbers

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 5;
    cout << "Sum: " << a + b << endl;
    return 0;
}
```

Output:

Sum: 15

Code 2: Subtraction of two numbers

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 5;
    cout << "Difference: " << a - b << endl;
    return 0;
```

```
}
```

Output:

Difference: 5

Code 3: Multiplication of two numbers

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 5;
    cout << "Product: " << a * b << endl;
    return 0;
}
```

Output:

Product: 50

Code 4: Division of two numbers

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 5;
    cout << "Quotient: " << a / b << endl;
    return 0;
}
```

Output:

Quotient: 2

Code 5: Modulus operation on two numbers

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 5;
    cout << "Remainder: " << a % b << endl;
    return 0;
}
```

Output:

Remainder: 0

Expt No. 2	CONTROL STRUCTURES	Date of Expt:

AIM:

Demonstrate control structures such as if-else, switch-case, for loop, while loop, and do-while loop.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Implement basic control flow statements.
2. Use different loops to display results.

Code 1: If-else condition

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 5;
   if (a > b) {
      cout << "a is greater than b" << endl;
   } else {
      cout << "b is greater than a" << endl;
   }
   return 0;
}
```

Output:

a is greater than b

Code 2: Switch-case statement

```
#include <iostream>
using namespace std;
int main() {
   int a = 3;
   switch (a) {
      case 1:
```

```
            cout << "One" << endl;
            break;
        case 2:
            cout << "Two" << endl;
            break;
        case 3:
            cout << "Three" << endl;
            break;
        default:
            cout << "Invalid number" << endl;
    }
    return 0;
}
```

Output:

Three

Code 3: For loop to print numbers

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
        cout << "Number: " << i << endl;
    }
    return 0;
}
```

Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

Code 4: While loop

```
#include <iostream>
```

```
using namespace std;
int main() {
    int i = 1;
    while (i <= 5) {
        cout << "Number: " << i << endl;
        i++;
    }
    return 0;
}
```

Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

Code 5: Do-while loop

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do {
        cout << "Number: " << i << endl;
        i++;
    } while (i <= 5);
    return 0;
}
```

Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

Expt No. 3	GRAPHIC LIBRARIES	Date of Expt:

AIM:

Create a simple graphical output using libraries like graphics.h.

SOFTWARE USED:

CodeBlocks (with a graphics library setup)

Procedure:

1. Initialize graphics mode.
2. Draw basic shapes like a circle, rectangle, and line.

Code 1: Drawing a circle

```
#include <graphics.h>
#include <conio.h>
using namespace std;
int main() {
   int gd = DETECT, gm;
   initgraph(&gd, &gm, "");
   circle(300, 200, 50); // Circle at (300, 200) with radius 50
   getch();
   closegraph();
   return 0;
}
```

Output: A circle drawn at the specified coordinates.

Code 2: Drawing a rectangle

```
#include <graphics.h>
#include <conio.h>
using namespace std;
int main() {
   int gd = DETECT, gm;
   initgraph(&gd, &gm, "");
```

```
    rectangle(100, 100, 400, 300); // Rectangle with top-left (100, 100) and bottom-right (400, 300)
    getch();
    closegraph();
    return 0;
}
```

Output: A rectangle drawn on the screen.

Code 3: Drawing a line

```
#include <graphics.h>
#include <conio.h>
using namespace std;
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");
    line(100, 100, 400, 100); // Line from (100, 100) to (400, 100)
    getch();
    closegraph();
    return 0;
}
```

Output: A horizontal line from (100, 100) to (400, 100).

Code 4: Drawing a rectangle with fill

```
#include <graphics.h>
#include <conio.h>
using namespace std;
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");
    setfillstyle(SOLID_FILL, BLUE);
    floodfill(150, 150, WHITE); // Fill the rectangle with blue
    rectangle(100, 100, 400, 300); // Rectangle
    getch();
```

```
    closegraph();
    return 0;
}
```

Output: A filled blue rectangle.

Code 5: Drawing multiple shapes

```
#include <graphics.h>
#include <conio.h>
using namespace std;
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");
    rectangle(100, 100, 400, 300);
    circle(250, 200, 50); // A circle inside the rectangle
    line(100, 100, 400, 300); // Diagonal line across the rectangle
    getch();
    closegraph();
    return 0;
}
```

Output: A rectangle with a circle inside and a diagonal line.

Expt No. 4	MATRIX MANIPULATION AND FUNCTIONS	Date of Expt:

AIM:

Perform basic matrix manipulations like addition, multiplication, and transposition.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Use loops to manipulate matrix elements.
2. Implement functions to add, multiply, and transpose matrices.

Code 1: Matrix Addition

```
#include <iostream>
using namespace std;
int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
    int sum[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum[i][j] = A[i][j] + B[i][j];
        }
    }
    cout << "Sum of matrices:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << sum[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
```

```
}
```

Output:

```
Sum of matrices:
6 8
10 12
```

Code 2: Matrix Multiplication

```
#include <iostream>
using namespace std;
int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
    int product[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            product[i][j] = 0;
            for (int k = 0; k < 2; k++) {
                product[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    cout << "Product of matrices:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << product[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

Product of matrices:

19 22

43 50

Code 3: Matrix Transposition

```
#include <iostream>
using namespace std;
int main() {
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int transpose[3][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            transpose[j][i] = A[i][j];
        }
    }
    cout << "Transpose of the matrix:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << transpose[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

Transpose of the matrix:

1 4

2 5

3 6

Code 4: Scalar Multiplication of Matrix

```
#include <iostream>
using namespace std;
int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int scalar = 2;
    int result[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = A[i][j] * scalar;
        }
    }
    cout << "Matrix after scalar multiplication:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
Matrix after scalar multiplication:
2 4
6 8
```

Code 5: Checking Matrix Symmetry

```
#include <iostream>
using namespace std;
int main() {
    int A[2][2] = {{1, 2}, {2, 1}};
```

```
    bool isSymmetric = true;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            if (A[i][j] != A[j][i]) {
                isSymmetric = false;
                break;
            }
        }
    }
    if (isSymmetric)
        cout << "The matrix is symmetric." << endl;
    else
        cout << "The matrix is not symmetric." << endl;
    return 0;
}
```

Output:

The matrix is symmetric.

Expt No. 5	OPERATOR OVERLOADING – BINARY AND UNARY OPERATORS AS FRIEND AND MEMBER FUNCTIONS	Date of Expt:

AIM:

Overload binary and unary operators to perform custom operations using friend and member functions.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Implement operator overloading for binary and unary operators.
2. Use friend functions for binary operators and member functions for unary operators.

Code 1: Unary Operator Overloading (Prefix)

```
#include <iostream>
using namespace std;
class Counter {
private:
   int count;
public:
   Counter() : count(0) {}
   Counter operator ++ () { // Prefix increment
      ++count;
      return *this;
   }
   void display() { cout << "Count: " << count << endl; }
};
int main() {
   Counter c;
   ++c;
   c.display();
   return 0;
```

```
}
```

Output:

```
Count: 1
```

Code 2: Unary Operator Overloading (Postfix)

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
    Counter() : count(0) {}
    Counter operator ++ (int) { // Postfix increment
        Counter temp = *this;
        count++;
        return temp;
    }
    void display() { cout << "Count: " << count << endl; }
};
int main() {
    Counter c;
    c++;
    c.display();
    return 0;
}
```

Output:

```
Count: 1
```

Code 3: Binary Operator Overloading using Friend Function

```
#include <iostream>
using namespace std;
class Box {
```

```
private:
    int length;
public:
    Box() : length(0) {}
    Box(int l) : length(l) {}
    friend Box operator + (Box b1, Box b2);
    void display() { cout << "Length: " << length << endl; }
};
Box operator + (Box b1, Box b2) { // Binary operator overloading
    Box temp;
    temp.length = b1.length + b2.length;
    return temp;
}
int main() {
    Box b1(5), b2(10);
    Box b3 = b1 + b2;
    b3.display();
    return 0;
}
```

Output:

Length: 15

Code 4: Overloading << Operator (Output Stream)

```
#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}
    friend ostream& operator << (ostream &out, Point p);
```

```
};
ostream& operator << (ostream &out, Point p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}
int main() {
    Point p(3, 4);
    cout << p << endl;
    return 0;
}
```

Output:

(3, 4)

Code 5: Overloading [] Operator (Array Access)

```
#include <iostream>
using namespace std;
class Array {
private:
    int arr[5];
public:
    Array() {
        for (int i = 0; i < 5; i++) arr[i] = i + 1;
    }
    int operator [] (int index) { return arr[index]; }
};
int main() {
    Array a;
    cout << "Element at index 2: " << a[2] << endl;
    return 0;
}
```

Output: Element at index 2: 3

Expt No. 6	UNARY OPERATORS - PREFIX AND POSTFIX FORM	Date of Expt:

AIM: Demonstrate prefix and postfix form of unary operator overloading.

SOFTWARE USED: CodeBlocks

Procedure:

1. Implement both prefix and postfix forms of unary operator overloading.

Code 1: Prefix Operator Overloading

```
#include <iostream>
using namespace std;
class Counter {
private:
   int count;
public:
   Counter() : count(0) {}
   Counter operator ++ () { // Prefix increment
      ++count;
      return *this;
   }
   void display() { cout << "Count: " << count << endl; }
};
int main() {
   Counter c;
   ++c;
   c.display();
   return 0;
}
```

Output:

```
Count: 1
```

Code 2: Postfix Operator Overloading

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
    Counter() : count(0) {}
    Counter operator ++ (int) { // Postfix increment
        Counter temp = *this;
        count++;
        return temp;
    }
    void display() { cout << "Count: " << count << endl; }
};
int main() {
    Counter c;
    c++;
    c.display();
    return 0;
}
```

Output:

Count: 1

Code 3: Combining Prefix and Postfix Operators

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
```

```
    Counter() : count(0) {}
    Counter operator ++ () { // Prefix increment
        ++count;
        return *this;
    }
    Counter operator ++ (int) { // Postfix increment
        Counter temp = *this;
        count++;
        return temp;
    }
    void display() { cout << "Count: " << count << endl; }
};
int main() {
    Counter c;
    ++c;
    c++;
    c.display();
    return 0;
}
```

Output:

Count: 2

Code 4: Postfix Increment with Value Return

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
    Counter() : count(0) {}
    int operator ++ (int) { // Postfix increment with return value
```

```
        int temp = count;
        count++;
        return temp;
    }
    void display() { cout << "Count: " << count << endl; }
};
int main() {
    Counter c;
    int temp = c++;
    cout << "Returned value: " << temp << endl;
    c.display();
    return 0;
}
```

Output:

Returned value: 0

Count: 1

Code 5: Prefix Decrement Operator

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
    Counter() : count(10) {}
    Counter operator -- () { // Prefix decrement
        --count;
        return *this;
    }
    void display() { cout << "Count: " << count << endl; }
};
```

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

Output:

```
Count: 9
```

Expt No. 7	NESTING OF MEMBER FUNCTIONS	Date of Expt:

AIM:

Demonstrate the nesting of member functions in C++.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Call a member function within another member function.

Code 1: Nested Function for Sum Calculation

```
#include <iostream>
using namespace std;
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
    void calculate() {
        int result = add(3, 4); // Nested call to add()
        cout << "Sum: " << result << endl;
    }
};
int main() {
    Calculator c;
    c.calculate();
    return 0;
}
```

Output:

Sum: 7

Code 2: Nested Function for Product Calculation

```
#include <iostream>
using namespace std;
class Calculator {
public:
    int multiply(int a, int b) {
        return a * b;
    }
    void calculate() {
        int result = multiply(5, 6); // Nested call to multiply()
        cout << "Product: " << result << endl;
    }
};
int main() {
    Calculator c;
    c.calculate();
    return 0;
}
```

Output:

```
Product: 30
```

Code 3: Factorial Calculation Using Nested Function

```
#include <iostream>
using namespace std;
class Factorial {
public:
    int fact(int n) {
        if (n == 0) return 1;
        return n * fact(n - 1);
    }
    void calculate() {
```

```
        int result = fact(5); // Nested call to fact()
        cout << "Factorial: " << result << endl;
    }
};
int main() {
    Factorial f;
    f.calculate();
    return 0;
}
```

Output:

Factorial: 120

Code 4: Nested Function for Checking Prime

```
#include <iostream>
using namespace std;
class Prime {
public:
    bool isPrime(int n) {
        if (n <= 1) return false;
        for (int i = 2; i < n; i++) {
            if (n % i == 0) return false;
        }
        return true;
    }
    void checkPrime() {
        int num = 7;
        if (isPrime(num)) {
            cout << num << " is a prime number." << endl;
        } else {
            cout << num << " is not a prime number." << endl;
        }
```

```
    }
};
int main() {
    Prime p;
    p.checkPrime();
    return 0;
}
```

Output:

7 is a prime number.

Code 5: Nested Function for Average Calculation

```
#include <iostream>
using namespace std;
class Average {
public:
    float avg(int a, int b) {
        return (a + b) / 2.0;
    }
    void calculate() {
        float result = avg(4, 6); // Nested call to avg()
        cout << "Average: " << result << endl;
    }
};
int main() {
    Average a;
    a.calculate();
    return 0;
}
```

Output:

Average: 5

Expt No. 8	CONSTRUCTORS AND DESTRUCTORS	Date of Expt:

AIM:

Demonstrate the usage of constructors and destructors in C++.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Use constructors to initialize objects and destructors to clean up resources.

Code 1: Simple Constructor and Destructor

```
#include <iostream>
using namespace std;
class Test {
public:
   Test() { cout << "Constructor called!" << endl; }
   ~Test() { cout << "Destructor called!" << endl; }
};
int main() {
   Test t;
   return 0;
}
```

Output:

Constructor called!

Destructor called!

Code 2: Parameterized Constructor

```
#include <iostream>
using namespace std;
class Box {
private:
   int length;
```

```
public:
    Box(int l) : length(l) { cout << "Box created with length: " << length << endl; }
    ~Box() { cout << "Box destroyed!" << endl; }
};
int main() {
    Box b(10);
    return 0;
}
```

Output:

```
Box created with length: 10
Box destroyed!
```

Code 3: Destructor for Resource Cleanup

```
#include <iostream>
using namespace std;
class FileHandler {
public:
    FileHandler() { cout << "File opened!" << endl; }
    ~FileHandler() { cout << "File closed!" << endl; }
};

int main() {
    FileHandler f;
    return 0;
}
```

Output:

```
File opened!
File closed!
```

Code 4: Default Constructor and Destructor

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

```
class Book {
public:
    Book() { cout << "Book created!" << endl; }
    ~Book() { cout << "Book destroyed!" << endl; }
};

int main() {
    Book b;
    return 0;
}
```

Output:

```
Book created!
Book destroyed!
```

Code 5: Constructor with Dynamic Memory Allocation

```
#include <iostream>
using namespace std;
class DynamicMemory {
private:
    int* ptr;
public:
    DynamicMemory(int size) {
        ptr = new int[size];
        cout << "Memory allocated!" << endl;
    }
    ~DynamicMemory() {
        delete[] ptr;
        cout << "Memory freed!" << endl;
    }
};
```

```
int main() {
    DynamicMemory dm(10);
    return 0;
}
```

Output:

```
Memory allocated!
Memory freed!
```

Expt No. 9	CONSTRUCTORS OVERLOADING	Date of Expt:

AIM:

Demonstrate constructor overloading in C++.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Define multiple constructors with different parameters.

Code 1: Constructor Overloading

```
#include <iostream>
using namespace std;
class Box {
private:
   int length, breadth, height;
public:
   Box() : length(0), breadth(0), height(0) {}
   Box(int l, int b, int h) : length(l), breadth(b), height(h) {}
   void display() { cout << "Dimensions: " << length << " x " << breadth << " x " << height << endl; }
};
int main() {
   Box b1;
   Box b2(5, 10, 15);
   b1.display();
   b2.display();
   return 0;
}
```

Output:

Dimensions: 0 x 0 x 0

Dimensions: 5 x 10 x 15

Code 2: Constructor with Default Parameters

```
#include <iostream>
using namespace std;
class Box {
private:
   int length;
public:
   Box(int l = 10) : length(l) {}
   void display() { cout << "Length: " << length << endl; }
};
int main() {
   Box b1;
   Box b2(20);
   b1.display();
   b2.display();
   return 0;
}
```

Output:

Length: 10

Length: 20

Code 3: Constructor Overloading with Different Arguments

```
#include <iostream>
using namespace std;
class Box {
private:
   int length;
public:
   Box() : length(0) {}
```

```
    Box(int l) : length(l) {}
    void display() { cout << "Length: " << length << endl; }
};
int main() {
    Box b1;
    Box b2(15);
    b1.display();
    b2.display();
    return 0;
}
```

Output:

```
Length: 0
Length: 15
```

Code 4: Overloading Constructors with Different Number of Parameters

```
#include <iostream>
using namespace std;
class Box {
private:
    int length, width;
public:
    Box() : length(0), width(0) {}
    Box(int l) : length(l), width(0) {}
    Box(int l, int w) : length(l), width(w) {}
    void display() { cout << "Length: " << length << ", Width: " << width << endl; }
};
int main() {
    Box b1;
    Box b2(5);
    Box b3(10, 20);
    b1.display();
```

```
    b2.display();
    b3.display();
    return 0;
}
```

Output:

Length: 0, Width: 0

Length: 5, Width: 0

Length: 10, Width: 20

Code 5: Constructor Overloading with Member Initialization

```
#include <iostream>
using namespace std;
class Box {
private:
    int length;
public:
    Box(int l = 1) : length(l) {}
    void display() { cout << "Length: " << length << endl; }
};
int main() {
    Box b1(10);
    Box b2;
    b1.display();
    b2.display();
    return 0;
}
```

Output:

Length: 10

Length: 1

Expt No. 10	INHERITANCE AND ITS FORMS	Date of Expt:

AIM:

Demonstrate inheritance in C++ with different types of inheritance.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Implement inheritance with base and derived classes.
2. Demonstrate single, multiple, and multilevel inheritance.

Code 1: Single Inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
   void speak() { cout << "Animal speaks!" << endl; }
};
class Dog : public Animal {
public:
   void bark() { cout << "Dog barks!" << endl; }
};
int main() {
   Dog d;
   d.speak(); // Inherited function
   d.bark();  // Derived class function
   return 0;
}
```

Output:

```
Animal speaks!
Dog barks!
```

Code 2: Multiple Inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
    void speak() { cout << "Animal speaks!" << endl; }
};
class Bird {
public:
    void fly() { cout << "Bird flies!" << endl; }
};
class Eagle : public Animal, public Bird {
public:
    void hunt() { cout << "Eagle hunts!" << endl; }
};
int main() {
    Eagle e;
    e.speak(); // Inherited from Animal
    e.fly();   // Inherited from Bird
    e.hunt();  // Derived class function
    return 0;
}
```

Output:

```
Animal speaks!
Bird flies!
Eagle hunts!
```

Code 3: Multilevel Inheritance

```
#include <iostream>
using namespace std;
class Animal {
```

```
public:
    void speak() { cout << "Animal speaks!" << endl; }
};
class Dog : public Animal {
public:
    void bark() { cout << "Dog barks!" << endl; }
};
class Poodle : public Dog {
public:
    void jump() { cout << "Poodle jumps!" << endl; }
};
int main() {
    Poodle p;
    p.speak();  // Inherited from Animal
    p.bark();   // Inherited from Dog
    p.jump();   // Derived class function
    return 0;
}
```

Output:

```
Animal speaks!
Dog barks!
Poodle jumps!
```

Code 4: Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
    void speak() { cout << "Animal speaks!" << endl; }
};
class Dog : public Animal {
```

```
public:
    void bark() { cout << "Dog barks!" << endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Cat meows!" << endl; }
};
int main() {
    Dog d;
    d.speak();  // Inherited from Animal
    d.bark();   // Derived class function
    Cat c;
    c.speak();  // Inherited from Animal
    c.meow();   // Derived class function
    return 0;
}
```

Output:

```
Animal speaks!
Dog barks!
Animal speaks!
Cat meows!
```

Code 5: Hybrid Inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
    void speak() { cout << "Animal speaks!" << endl; }
};
class Bird {
public:
```

```
    void fly() { cout << "Bird flies!" << endl; }
};
class Bat : public Animal, public Bird {
public:
    void hang() { cout << "Bat hangs!" << endl; }
};
int main() {
    Bat b;
    b.speak();  // Inherited from Animal
    b.fly();    // Inherited from Bird
    b.hang();   // Derived class function
    return 0;
}
```

Output:

```
Animal speaks!
Bird flies!
Bat hangs!
```

Expt No. 11	VISIBILITY MODE – PUBLIC, PRIVATE, AND PROTECTED	Date of Expt:

AIM: Demonstrate the use of public, private, and protected access specifiers in C++.

SOFTWARE USED: CodeBlocks

Procedure:

1. Implement different access specifiers (public, private, and protected) to control member visibility.

Code 1: Public Access Specifier

```
#include <iostream>
using namespace std;
class Box {
public:
   int length;
   Box() : length(10) {}
   void display() { cout << "Length: " << length << endl; }
};
int main() {
   Box b;
   cout << "Length: " << b.length << endl;
   b.display();
   return 0;
}
```

Output:

Length: 10

Length: 10

Code 2: Private Access Specifier

```
#include <iostream>
using namespace std;
class Box {
private:
```

```
    int length;
public:
    Box() : length(10) {}
    void display() { cout << "Length: " << length << endl; }
};
int main() {
    Box b;
    // cout << "Length: " << b.length << endl; // Error: Cannot access private member
    b.display();
    return 0;
}
```

Output:

Length: 10

Code 3: Protected Access Specifier

```
#include <iostream>
using namespace std;
class Box {
protected:
    int length;
public:
    Box() : length(10) {}
};
class Derived : public Box {
public:
    void display() { cout << "Length: " << length << endl; }
};
int main() {
    Derived d;
    d.display();
    return 0;
```

```
}
```

Output:

```
Length: 10
```

Code 4: Private and Public Members

```
#include <iostream>
using namespace std;
class Box {
private:
   int length;
public:
   Box() : length(10) {}
   void display() { cout << "Length: " << length << endl; }
};
int main() {
   Box b;
   // cout << b.length; // Error: Cannot access private member
   b.display();
   return 0;
}
```

Output:

```
Length: 10
```

Code 5: Using Protected for Inheritance

```
#include <iostream>
using namespace std;
class Base {
protected:
   int value;
public:
   Base() : value(10) {}
};
```

```
class Derived : public Base {
public:
    void display() { cout << "Value: " << value << endl; }
};
int main() {
    Derived d;
    d.display();
    return 0;
}
```

Output:

Value: 10

Expt No. 12	RUNTIME POLYMORPHISM – VIRTUAL FUNCTIONS	Date of Expt:

AIM:

Demonstrate runtime polymorphism using virtual functions in C++.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Implement virtual functions to achieve runtime polymorphism.

Code 1: Basic Virtual Function Example

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void display() { cout << "Base class" << endl; }
};
class Derived : public Base {
public:
    void display() override { cout << "Derived class" << endl; }
};
int main() {
    Base* ptr;
    Derived d;
    ptr = &d;
    ptr->display(); // Calls Derived class function
    return 0;
}
```

Output:

Derived class

Code 2: Virtual Function with Inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() { cout << "Animal speaks!" << endl; }
};
class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks!" << endl; }
};
int main() {
    Animal* ptr;
    Dog d;
    ptr = &d;
    ptr->speak(); // Calls Dog class function
    return 0;
}
```

Output:

```
Dog barks!
```

Code 3: Virtual Destructor Example

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual ~Base() { cout << "Base Destructor" << endl; }
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor" << endl; }
```

```
};
int main() {
    Base* ptr = new Derived;
    delete ptr; // Calls virtual destructors
    return 0;
}
```

Output:

```
Derived Destructor
Base Destructor
```

Code 4: Virtual Function with Different Return Types

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual int add(int a, int b) { return a + b; }
};
class Derived : public Base {
public:
    int add(int a, int b) override { return a * b; }
};
int main() {
    Base* ptr = new Derived;
    cout << "Result: " << ptr->add(3, 4) << endl; // Calls Derived class function
    delete ptr;
    return 0;
}
```

Output:

```
Result: 12
```

Code 5: Virtual Functions with Multiple Inheritance

```
#include <iostream>
using namespace std;
class A {
public:
    virtual void show() { cout << "Class A" << endl; }
};
class B {
public:
    virtual void display() { cout << "Class B" << endl; }
};
class C : public A, public B {
public:
    void show() override { cout << "Class C (show)" << endl; }
    void display() override { cout << "Class C (display)" << endl; }
};
int main() {
    C c;
    A* a = &c;
    B* b = &c;
    a->show();
    b->display();
    return 0;
}
```

Output:

```
Class C (show)
Class C (display)
```

Expt No. 13	FILE OPENING AND FILE CLOSING	Date of Expt:

AIM:

Demonstrate file opening, writing, and closing in C++.

SOFTWARE USED:

CodeBlocks

Procedure:

1. Open a file for writing, write to it, and then close the file.

Code 1: File Writing

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream outfile("example.txt");
    outfile << "Hello, file!" << endl;
    outfile.close();
    return 0;
}
```

Output:

The file "example.txt" will contain the text:

Hello, file!

Code 2: File Reading

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream infile("example.txt");
    string line;
    while (getline(infile, line)) {
        cout << line << endl;
```

```
    }
    infile.close();
    return 0;
}
```

Output:

Hello, file!

Code 3: File Opening Error Handling

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream infile("nonexistent.txt");
    if (!infile) {
        cout << "Error opening file!" << endl;
    }
    infile.close();
    return 0;
}
```

Output:

Error opening file!

Code 4: File Appending

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream outfile("example.txt", ios::app);
    outfile << "Appending new line!" << endl;
    outfile.close();
    return 0;
}
```

Output:
The file "example.txt" will contain:

Hello, file!

Appending new line!

Code 5: File Write and Read Together

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream outfile("example.txt");
    outfile << "Writing to the file!" << endl;
    outfile.close();
    ifstream infile("example.txt");
    string line;
    while (getline(infile, line)) {
        cout << line << endl;
    }
    infile.close();
    return 0;
}
```

Output:

Writing to the file