Name – Anish
UID – 22BCS15292
Section – 620-A

DAY 1

Very Easy

1) Sum of Natural Numbers up to N

Calculate the sum of all natural numbers from 1 to n, where n is a positive integer. Use the formula:
Sum=n×(n+1)/2 .
Take n as input and output the sum of natural numbers from 1 to n .

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

int main() {
 int n;
 cout << "Enter a positive integer: ";
 cin >> n;

 int sum = n * (n + 1) / 2;
 cout << "The sum of natural numbers from 1 to " << n << " is: " << sum << endl;

 return 0;
}
```

2) Check if a Number is Prime

Objective

Check if a given number n is a prime number. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

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

bool isPrime(int n) {
 if (n <= 1) return false;
 for (int i = 2; i * i <= n; i++) {
 if (n % i == 0) return false;
```

```
 }
 return true;
}

int main() {
 int n;
 cout << "Enter a number: ";
 cin >> n;

 if (isPrime(n))
 cout << n << " is a prime number." << endl;
 else
 cout << n << " is not a prime number." << endl;

 return 0;
}
```

Easy:

1) Count Digits in a Number

Objective

Count the total number of digits in a given number n. The number can be a positive integer. For example, for the number 12345, the count of digits is 5. For a number like 900000, the count of digits is 6.

Given an integer n, your task is to determine how many digits are present in n. This task will help you practice working with loops, number manipulation, and conditional logic.

```
 //no. of digits
#include<iostream>
Using namespace std;
Int main(){
Int b;
Cin>>b;
 int ans = 0;
 while(b>0){
 ans++;
 b/=10;
 }
 cout<<ans<<endl;
```

}

2) Reverse a Number

Objective

Reverse the digits of a given number n. For example, if the input number is 12345, the output should be 54321. The task involves using loops and modulus operators to extract the digits and construct the reversed number.

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

int main() {
 int n, reversed = 0;
 cout << "Enter a number: ";
 cin >> n;

 while (n != 0) {
 int digit = n % 10;
 reversed = reversed * 10 + digit;
 n /= 10;
 }

 cout << "Reversed number: " << reversed << endl;
 return 0;
}
```

Medium:

1) **Function Overloading for Calculating Area.**

Objective

Write a program to calculate the area of different shapes using function overloading. Implement overloaded functions to compute the area of a circle, a rectangle, and a triangle.

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

// Function to calculate the area of a circle
double area(double radius) {
 return 3.14159 * radius * radius;
}
```

```
// Function to calculate the area of a rectangle
double area(double length, double width) {
 return length * width;
}

// Function to calculate the area of a triangle
double area(double base, double height, bool isTriangle) {
 return 0.5 * base * height;
}

int main() {
 double radius, length, width, base, height;

 cout << "Enter the radius of the circle: ";
 cin >> radius;
 cout << "Area of the circle: " << area(radius) << endl;

 cout << "Enter the length and width of the rectangle: ";
 cin >> length >> width;
 cout << "Area of the rectangle: " << area(length, width) << endl;

 cout << "Enter the base and height of the triangle: ";
 cin >> base >> height;
 cout << "Area of the triangle: " << area(base, height, true) << endl;

 return 0;
}
```

2) Func(on Overloading with Hierarchical Structure.

Objec&ve

Write a program that demonstrates function overloading to calculate the salary of employees at different levels in a company hierarchy. Implement overloaded functions to compute salary for:
- Intern (basic stipend).
- Regular employee (base salary + bonuses).
- Manager (base salary + bonuses + performance incentives).

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

// Function to calculate salary for an intern
double calculateSalary(double stipend) {
 return stipend;
}
```

```
// Function to calculate salary for a regular employee
double calculateSalary(double baseSalary, double bonuses) {
 return baseSalary + bonuses;
}

// Function to calculate salary for a manager
double calculateSalary(double baseSalary, double bonuses, double incentives)
{  return baseSalary + bonuses + incentives;
}

int main() {
 double stipend, baseSalary, bonuses, incentives;

 // Calculate salary for an intern
 cout << "Enter the stipend for the intern: ";
 cin >> stipend;
 cout << "Salary of the intern: " << calculateSalary(stipend) << endl;

 // Calculate salary for a regular employee
 cout << "Enter the base salary and bonuses for the regular employee: ";
cin >> baseSalary >> bonuses;
 cout << "Salary of the regular employee: " << calculateSalary(baseSalary, bonuses) << endl;

 // Calculate salary for a manager
 cout << "Enter the base salary, bonuses, and incentives for the manager: ";
cin >> baseSalary >> bonuses >> incentives;
 cout << "Salary of the manager: " << calculateSalary(baseSalary, bonuses, incentives) <<
endl;

 return 0;
}
```

3) Encapsula(on with Employee Details

Objec&ve

Write a program that demonstrates encapsulation by creating a class Employee. The class should have private attributes to store:

Employee ID.

Employee Name.

Employee Salary.

Provide public methods to set and get these attributes, and a method to display all details of the employee.

```
#include <iostream>
```

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

class Employee {
private:
 int employeeID;
 string employeeName;
 float employeeSalary;

public:
 // Setter methods
 void setEmployeeID(int id) {
 if (id >= 1 && id <= 1000000) {
 employeeID = id;
 } else {
 cout << "Invalid Employee ID!" << endl;  }
 }

 void setEmployeeName(const string& name) {
if (name.length() <= 50) {
 employeeName = name;
 } else {
 cout << "Name length exceeds 50 characters!" << endl;  }
 }

 void setEmployeeSalary(float salary) {
 if (salary >= 1.0 && salary <= 10000000.0) {
employeeSalary = salary;
 } else {
 cout << "Invalid Salary!" << endl;
 }
 }

 // Getter methods
 int getEmployeeID() const {
 return employeeID;
 }

 string getEmployeeName() const {
 return employeeName;
 }

 float getEmployeeSalary() const {
 return employeeSalary;
 }
```

```
// Method to display employee details
void displayDetails() const {
cout << "Employee ID: " << employeeID << endl;
cout << "Employee Name: " << employeeName << endl;
cout << "Employee Salary: " << employeeSalary << endl;
}
};

int main() {
Employee emp;
int id;
string name;
float salary;

cout << "Enter Employee ID: ";
cin >> id;
cin.ignore(); // Clear the newline from input buffer

cout << "Enter Employee Name: ";
getline(cin, name);

cout << "Enter Employee Salary: ";
cin >> salary;

// Set attributes using setters
emp.setEmployeeID(id);
emp.setEmployeeName(name);
emp.setEmployeeSalary(salary);

// Display employee details
cout << "\nEmployee Details:" << endl;
emp.displayDetails();

return 0;
}
```

4) Inheritance with Student and Result Classes.

Objec&ve

Create a program that demonstrates inheritance by defining:

- A base class Student to store details like Roll Number and Name. - A derived class Result to store marks for three subjects and calculate the total and percentage.

```
#include <iostream>
```

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

// Base class Student
class Student {
protected:
 int rollNumber;
 string name;

public:
 void setDetails(int roll, const string& studentName) {
rollNumber = roll;
 name = studentName;
 }

 void displayDetails() const {
 cout << "Roll Number: " << rollNumber << endl;
cout << "Name: " << name << endl;
 }
};

// Derived class Result
class Result : public Student {
private:
 int marks[3];
 int total;
 float percentage;

public:
 void setMarks(int mark1, int mark2, int mark3) {
marks[0] = mark1;
 marks[1] = mark2;
 marks[2] = mark3;
 }

 void calculateResult() {
 total = marks[0] + marks[1] + marks[2];
percentage = total / 3.0;
 }

 void displayResult() const {
  cout << "Marks in Subject 1: " << marks[0] << endl;
cout << "Marks in Subject 2: " << marks[1] << endl;
cout << "Marks in Subject 3: " << marks[2] << endl;
cout << "Total Marks: " << total << endl;
```

```
 cout << "Percentage: " << percentage << "%" << endl;  }
};

int main() {
 Result student;
 int roll, mark1, mark2, mark3;
 string name;

 // Input student details
 cout << "Enter Roll Number: ";
 cin >> roll;
 cin.ignore(); // Clear the newline from input buffer

 cout << "Enter Name: ";
 getline(cin, name);

 cout << "Enter Marks in 3 Subjects: ";
 cin >> mark1 >> mark2 >> mark3;

 // Set details and marks
 student.setDetails(roll, name);
 student.setMarks(mark1, mark2, mark3);

 // Calculate and display result
 student.calculateResult();

 cout << "\nStudent Details and Result:" << endl;
 student.displayDetails();
 student.displayResult();

 return 0;
}
```

5) Polymorphism with Shape Area Calcula(on.

Objec&ve

Create a program that demonstrates polymorphism by calculating the area of different shapes using a base class Shape and derived classes for Circle, Rectangle, and Triangle. Each derived class should override a virtual function to compute the area of the respective shape.

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

// Base class Shape
class Shape {
```

```
public:
 virtual void calculateArea() const = 0; // Pure virtual function
 virtual ~Shape() {} // Virtual destructor
};
// Derived class Circle
class Circle : public Shape {
private:
 double radius;

public:
 Circle(double r) : radius(r) {}

 void calculateArea() const override {
 double area = M_PI * radius * radius;
 cout << "Area of the Circle: " << area << endl;  }
};

// Derived class Rectangle
class Rectangle : public Shape {
private:
 double length, breadth;

public:
 Rectangle(double l, double b) : length(l), breadth(b) {}

 void calculateArea() const override {
 double area = length * breadth;
 cout << "Area of the Rectangle: " << area << endl;  }
};

// Derived class Triangle
class Triangle : public Shape {
private:
 double base, height;

public:
 Triangle(double b, double h) : base(b), height(h) {}

 void calculateArea() const override {
 double area = 0.5 * base * height;
 cout << "Area of the Triangle: " << area << endl;  }
};

int main() {
```

```
 double radius, length, breadth, base, height;

// Input for Circle

 cout << "Enter the radius of the circle: ";
 cin >> radius;
 Circle circle(radius);

 // Input for Rectangle
 cout << "Enter the length and breadth of the rectangle: ";
 cin >> length >> breadth;
 Rectangle rectangle(length, breadth);

 // Input for Triangle
 cout << "Enter the base and height of the triangle: ";
 cin >> base >> height;
 Triangle triangle(base, height);

 // Polymorphic behavior
 Shape* shapes[] = {&circle, &rectangle, &triangle};

 cout << "\nCalculating Areas:" << endl;
 for (Shape* shape : shapes) {
 shape->calculateArea();
 }

 return 0;
}
```

Hard:

1)Implemen(ng Polymorphism for Shape Hierarchies.

Objec&ve

Write a program to demonstrate runtime polymorphism in C++ using a base class Shape and derived classes Circle, Rectangle, and Triangle. The program should use virtual functions to calculate and print the area of each shape based on user input.

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

// Base class Shape
class Shape {
public:
 virtual void inputDimensions() = 0; // Pure virtual function for input  virtual void
```

```
calculateArea() const = 0; // Pure virtual function for area calculation  virtual
~Shape() {} // Virtual destructor
};

// Derived class Circle
class Circle : public Shape {
private:
 double radius;

public:
 void inputDimensions() override {
 cout << "Enter the radius of the circle: ";
 cin >> radius;
 }

 void calculateArea() const override {
 double area = M_PI * radius * radius;
 cout << "Area of the Circle: " << area << endl;  }
};

// Derived class Rectangle
class Rectangle : public Shape {
private:
 double length, breadth;

public:
 void inputDimensions() override {
 cout << "Enter the length and breadth of the rectangle: ";
cin >> length >> breadth;
 }

 void calculateArea() const override {
 double area = length * breadth;
 cout << "Area of the Rectangle: " << area << endl;  }
};

// Derived class Triangle
class Triangle : public Shape {
private:
 double base, height;

public:
 void inputDimensions() override {
 cout << "Enter the base and height of the triangle: ";
cin >> base >> height;
```

```
}

void calculateArea() const override {
double area = 0.5 * base * height;
cout << "Area of the Triangle: " << area << endl;
}
};

int main() {
Shape* shape = nullptr;
int choice;

cout << "Choose a shape to calculate the area:\n";
cout << "1. Circle\n2. Rectangle\n3. Triangle\n";
cout << "Enter your choice: ";
cin >> choice;

switch (choice) {
case 1:
shape = new Circle();
break;
case 2:
shape = new Rectangle();
break;
case 3:
shape = new Triangle();
break;
default:
cout << "Invalid choice!" << endl;
return 1;
}

shape->inputDimensions();
shape->calculateArea();

delete shape; // Free allocated memory
return 0;
}
```

2) Matrix Mul(plica(on Using Func(on Overloading

Objec&ve

Implement matrix operations in C++ using function overloading. Write a function operate() that can perform:

- **Matrix Addition** for matrices of the same dimensions.

- **Matrix Multiplication** where the number of columns of the first matrix equals the number of rows of the second matrix.

```
#include <iostream>
#include <vector>
using namespace std;
class Matrix {
private:
 vector<vector<int>> mat;
 int rows, cols;

public:
 Matrix(int r, int c) : rows(r), cols(c) {
 mat.resize(r, vector<int>(c, 0));
 }

 void inputMatrix() {
 cout << "Enter elements of the matrix (" << rows << "x" << cols << "):" << endl;  for
(int i = 0; i < rows; ++i) {
 for (int j = 0; j < cols; ++j) {
 cin >> mat[i][j];
 }
 }
 }

 void displayMatrix() const {
 for (const auto& row : mat) {
 for (int elem : row) {
 cout << elem << " ";
 }
 cout << endl;
 }
 }

 // Matrix addition
 Matrix operate(const Matrix& other) const {
 if (rows != other.rows || cols != other.cols) {
 throw invalid_argument("Matrices must have the same dimensions for addition.");  }

 Matrix result(rows, cols);
 for (int i = 0; i < rows; ++i) {
 for (int j = 0; j < cols; ++j) {
 result.mat[i][j] = mat[i][j] + other.mat[i][j];
 }
 }
 return result;
```

```
 }

 // Matrix multiplication
 Matrix operate(const Matrix& other, bool multiply) const {
 if (cols != other.rows) {
 throw invalid_argument("Number of columns of the first matrix must equal number of  rows
of the second matrix.");
 }

 Matrix result(rows, other.cols);
 for (int i = 0; i < rows; ++i) {
 for (int j = 0; j < other.cols; ++j) {
 for (int k = 0; k < cols; ++k) {
 result.mat[i][j] += mat[i][k] * other.mat[k][j];
 }
 }
 }
 return result;
 }
};

int main() {
 int r1, c1, r2, c2;
 cout << "Enter dimensions of the first matrix (rows and columns): ";
 cin >> r1 >> c1;

 cout << "Enter dimensions of the second matrix (rows and columns): ";
cin >> r2 >> c2;

 Matrix mat1(r1, c1), mat2(r2, c2);

 cout << "\nMatrix 1:\n";
 mat1.inputMatrix();

 cout << "\nMatrix 2:\n";
 mat2.inputMatrix();

 try {
 cout << "\nMatrix Addition:" << endl;
 if (r1 == r2 && c1 == c2) {
 Matrix addition = mat1.operate(mat2);
 addition.displayMatrix();
 } else {
 cout << "Addition not possible due to different dimensions." << endl;  }
```

```
cout << "\nMatrix Multiplication:" << endl;
if (c1 == r2) {
Matrix multiplication = mat1.operate(mat2, true);
multiplication.displayMatrix();
} else {
cout << "Multiplication not possible due to dimension mismatch." << endl;  }
} catch (const exception& e) {
cout << "Error: " << e.what() << endl;
}

return 0;
}
```