Name – Aryan
UID – 22BCS15357
Section – 620(B)

Very Easy

1) Sum of Natural Numbers up to N

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

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

    int sum = n * (n + 1) / 2;
    cout << " sum : " << sum << endl;

    return 0;
}
```

2) Check if a Number is Prime

Objective

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

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

```
    return true;
}

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

    if (isPrime(n))
        cout << n << " prime << endl;
    else
        cout << n << " not prime" << endl;

    return 0;
}
```

3) Count Digits in a Number

```
 #include<iostream>
Using namespace std;
Int main(){
Int b;
Cin>>b;
    int ans = 0;
    while(b>0){
        ans++;
        b/=10;
    }
    cout<<ans<<endl;
```

Medium:

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

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

double area(double radius) {
    return 3.14159 * radius * radius;
}
double area(double length, double width) {
    return length * width;
}

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) Function Overloading with Hierarchical Structure.

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

double calculateSalary(double stipend) {
    return stipend;
}

double calculateSalary(double baseSalary, double bonuses) {
    return baseSalary + bonuses;
}


double calculateSalary(double baseSalary, double bonuses, double incentives) {
    return baseSalary + bonuses + incentives;
}

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

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

    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) Encapsulation with Employee Details

Objective

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:

    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;
```

```
        }
    }

    int getEmployeeID() const {
        return employeeID;
    }

    string getEmployeeName() const {
        return employeeName;
    }

    float getEmployeeSalary() const {
        return employeeSalary;
    }

    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();

    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;
}
```

Hard:

1) Implementing Polymorphism for Shape Hierarchies.

Objective

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;

class Shape {
public:
    virtual void inputDimensions() = 0;
    virtual void calculateArea() const = 0;
    virtual ~Shape() {}
};


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;
    }
};
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;
    }
};
```

```
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 Multiplication Using Function Overloading

Objective

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 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;
}
```

3) Create a C++ program to simulate a vehicle hierarchy using multi-level inheritance. Design a base class Vehicle that stores basic details (brand, model, and mileage). Extend it into the Car class to add attributes like fuel efficiency and speed. Further extend it into ElectricCar to include battery capacity and charging time. Implement methods to calculate:
Fuel Efficiency: Miles per gallon (for Car).

Range: Total distance the electric car can travel with a full charge.
Code:

```
#include <iostream>
#include <iomanip>
using namespace std;
class Vehicle {
protected:
string brand, model;
double mileage;
public:
Vehicle(string b, string m, double mi) : brand(b), model(m), mileage(mi) {}
void displayInfo() {
cout << "Vehicle: " << brand << " " << model << endl;
cout << "Mileage: " << mileage << endl;
}
};

class Car : public Vehicle {
protected:
double fuel, distance;
public:
Car(string b, string m, double mi, double f, double d) : Vehicle(b, m, mi), fuel(f),
distance(d) {}
double calculateFuelEfficiency() { return distance / fuel; }
};
class ElectricCar : public Car {
double batteryCapacity, efficiency;
public:
ElectricCar(string b, string m, double mi, double bc, double eff)
: Car(b, m, mi, 0, 0), batteryCapacity(bc), efficiency(eff) {}
double calculateRange() { return batteryCapacity * efficiency; }
};
int main() {
int type;
```

```
cout << "Vehicle Type (1 for Car, 2 for Electric Car): ";
cin >> type;
string brand, model;
double mileage;
cout << "Enter Brand, Model, and Mileage: ";
cin >> brand >> model >> mileage;
if (type == 1) { // Car
double fuel, distance;
cout << "Enter Fuel (gallons) and Distance (miles): ";
cin >> fuel >> distance;
Car car(brand, model, mileage, fuel, distance);
car.displayInfo();

cout << "Fuel Efficiency: " << fixed << setprecision(2) << car.calculateFuelEfficiency()
<< " miles/gallon" << endl;
}
else if (type == 2) { // ElectricCar
double batteryCapacity, efficiency;
cout << "Enter Battery Capacity (kWh) and Efficiency (miles per kWh): ";
cin >> batteryCapacity >> efficiency;
ElectricCar ecar(brand, model, mileage, batteryCapacity, efficiency);
ecar.displayInfo();
cout << "Range: " << fixed << setprecision(2) << ecar.calculateRange() << " miles";
}
else {
cout << "Invalid vehicle type!" << endl;
}
return 0;
}
```