

Name: Kaushik Dhruv Alok

UID: 22BCS10210

Section/Group: KPIT-901/A

Assignment - 1

Q1. Sum of Natural Numbers up to N. (V Easy) :

Calculate the sum of all natural numbers from 1 to n, where n is a positive integer. Use the formula:

$Sum = n \times (n+1) / 2$.

Take n as input and output the sum of natural numbers from 1 to n .

Q2. Count Digits in a Number. (Easy) :

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.

Q3. Function Overloading for Calculating Area. (Medium)

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.

Code:

```
#include <iostream>
```

```
#include <math.h>
```

```
using namespace std;
```

```
#define M_PI 3.14159265358979323846
```

```
int sumNaturalNum();
```

```
int sumNaturalNum(int n);
```

```
int countDigits();
```

```
int countDigits(int num);
```

```
float calcCircleArea();
```

```
float calcCircleArea(float rad);
```

```
float calcRectangleArea();
```

```
float calcRectangleArea(float length, float width);
```

```

float calcTriangleArea();
float calcTriangleArea(float base, float height);

int main(int argc, char const *argv[])
{
    cout << "Sum of natural numbers: " << sumNaturalNum() << endl;
    cout << "Number of digits: " << countDigits() << endl;
    cout << "Circle area: " << calcCircleArea() << endl;
    cout << "Rectangle area: " << calcRectangleArea() << endl;
    cout << "Triangle area: " << calcTriangleArea() << endl;

    return 0;
}

int sumNaturalNum(int n){
    int sum = n * (n + 1) / 2;
    return sum;
}

int sumNaturalNum(){
    int n;
    cout << "Enter a natural number: ";
    cin >> n;
    return sumNaturalNum(n);
}

int countDigits(int num){
    int digitCount = 0;
    while (num != 0)
    {
        num = num / 10;
        digitCount++;
    }
    return digitCount;
}

int countDigits(){
    int num;
    cout << "Enter an integer: ";
    cin >> num;
    return countDigits(num);
}

float calcCircleArea(float rad){
    float area = M_PI * pow(rad, 2);
    return area;
}

```

```
float calcCircleArea(){  
    float rad;  
    cout << "Enter radius of the circle: ";  
    cin >> rad;  
    return calcCircleArea(rad);  
}
```

```
float calcRectangleArea(float length, float width){  
    return length * width;  
}
```

```
float calcRectangleArea(){  
    float length, width;  
    cout << "Enter length and width of the rectangle: ";  
    cin >> length >> width;  
    return calcRectangleArea(length, width);  
}
```

```
float calcTriangleArea(float base, float height){  
    return 0.5f * base * height;  
}
```

```
float calcTriangleArea(){  
    float base, height;  
    cout << "Enter base and height of the triangle: ";  
    cin >> base >> height;  
    return calcTriangleArea(base, height);  
}
```

Output:

```
● PS C:\Users\Dhruv\winterCamp> & 'c:\Users\Dhruv\.vscode\ex
rosoft-MIEngine-In-cd4hweo0.4o3' '--stdout=Microsoft-MIEngi
aa2oe.1dh' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--inte
Sum of natural numbers: Enter a natural number: 10
55
Number of digits: Enter an integer: 255
3
Circle area: Enter radius of the circle: 4
50.2655
Rectangle area: Enter length and width of the rectangle: 5
10
50
Triangle area: Enter base and height of the triangle: 10
20
100
```

Q4. Design a C++ program to simulate a banking system using polymorphism. Create a base class Account with a virtual method calculateInterest(). Use the derived classes SavingsAccount and CurrentAccount to implement specific interest calculation logic:

- **SavingsAccount:** Interest = Balance × Rate × Time.
- **CurrentAccount:** No interest, but includes a maintenance fee deduction.

Code:

```
#include <iostream>
#include <memory>
#include <stdexcept>

class Account {
public:
    virtual ~Account() = default;
    virtual void calculateInterest() = 0; // Pure virtual method
};

class SavingsAccount : public Account {
private:
    int balance;
    double interestRate;
    int time; // in years
public:
    SavingsAccount(int bal, double rate, int t) : balance(bal), interestRate(rate), time(t) {}
    void calculateInterest() override {
        // Interest = Balance × (Rate/100) × Time
    }
};
```

```

        double interest = balance * (interestRate / 100.0) * time;
        std::cout << "Savings Account Interest: " << static_cast<int>(interest) << std::endl;
    }
};

class CurrentAccount : public Account {
private:
    int balance;
    int maintenanceFee;
public:
    CurrentAccount(int bal, int fee) : balance(bal), maintenanceFee(fee) {}
    void calculateInterest() override {
        // No interest for Current Account; Deduct maintenance fee.
        int finalBalance = balance - maintenanceFee;
        std::cout << "Balance after fee deduction: " << finalBalance << std::endl;
    }
};

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int accountType;
    if (!(std::cin >> accountType) || accountType < 1 || accountType > 2) {
        std::cout << "Invalid account type." << std::endl;
        return 0;
    }

    try {
        std::unique_ptr<Account> account;

        if (accountType == 1) {
            // Savings Account
            int balance;
            double interestRate;
            int time;
            if (!(std::cin >> balance >> interestRate >> time)) {
                std::cout << "Invalid input." << std::endl;
                return 0;
            }
            // Validate constraints
            if (balance < 1000 || balance > 1000000
                || interestRate < 1 || interestRate > 15
                || time < 1 || time > 10) {
                std::cout << "Invalid input." << std::endl;
                return 0;
            }

            account = std::make_unique<SavingsAccount>(balance, interestRate, time);
        } else {
            // Current Account

```

```

    int balance;
    int fee;
    if (!(std::cin >> balance >> fee)) {
        std::cout << "Invalid input." << std::endl;
        return 0;
    }
    // Validate constraints
    if (balance < 1000 || balance > 1000000
        || fee < 50 || fee > 500) {
        std::cout << "Invalid input." << std::endl;
        return 0;
    }

    account = std::make_unique<CurrentAccount>(balance, fee);
}

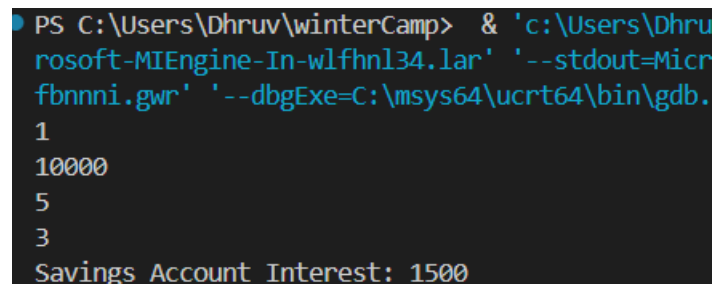
account->calculateInterest();

} catch (...) {
    std::cout << "Invalid input." << std::endl;
}

return 0;
}

```

Output:



```

PS C:\Users\Dhruv\winterCamp> & 'c:\Users\Dhruv\rossoft-MIEngine-In-wlfhnl34.lar' '--stdout=Microsoft.PowerShell.ConsoleHost\1.0.0.0\Microsoft.PowerShell.ConsoleHost.exe' --dbgExe=C:\msys64\ucrt64\bin\gdb.
1
10000
5
3
Savings Account Interest: 1500

```

Q5. Hierarchical Inheritance for Employee Management System

Create a C++ program to simulate an employee management system using hierarchical inheritance. Design a base class Employee that stores basic details (name, ID, and salary). Create two derived classes:

- **Manager:** Add and calculate bonuses based on performance ratings.
- **Developer:** Add and calculate overtime compensation based on extra hours worked.

The program should allow input for both types of employees and display their total earnings.

Code :

```

#include <iostream>
#include <string>
#include <memory>

class Employee {
protected:
    std::string name;
    int id;
    int salary;

public:
    Employee(const std::string &n, int i, int s) : name(n), id(i), salary(s) {}
    virtual ~Employee() = default;

    virtual void displayEarnings() const = 0; // pure virtual function
};

class Manager : public Employee {
private:
    int rating; // 1 to 5
public:
    Manager(const std::string &n, int i, int s, int r)
        : Employee(n, i, s), rating(r) {}

    void displayEarnings() const override {
        // Bonus per rating point = 10% of salary
        // total bonus = rating * 10% * salary = rating * (0.1 * salary)
        int bonus = static_cast<int>(rating * (0.1 * salary));
        int totalEarnings = salary + bonus;

        std::cout << "Employee: " << name << " (ID: " << id << ")\n";
        std::cout << "Role: Manager\n";
        std::cout << "Base Salary: " << salary << "\n";
        std::cout << "Bonus: " << bonus << "\n";
        std::cout << "Total Earnings: " << totalEarnings << "\n";
    }
};

class Developer : public Employee {
private:
    int extraHours; // 0 to 100
public:
    Developer(const std::string &n, int i, int s, int hours)
        : Employee(n, i, s), extraHours(hours) {}

    void displayEarnings() const override {
        // Overtime rate = $500 per hour
        int overtimeCompensation = extraHours * 500;
        int totalEarnings = salary + overtimeCompensation;

        std::cout << "Employee: " << name << " (ID: " << id << ")\n";
    }
};

```

```

        std::cout << "Role: Developer\n";
        std::cout << "Base Salary: " << salary << "\n";
        std::cout << "Overtime Compensation: " << overtimeCompensation << "\n";
        std::cout << "Total Earnings: " << totalEarnings << "\n";
    }
};

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int employeeType;
    if (!(std::cin >> employeeType) || employeeType < 1 || employeeType > 2) {
        std::cout << "Invalid employee type.\n";
        return 0;
    }

    std::string name;
    int id;
    int salary;
    if (!(std::cin >> name >> id >> salary)) {
        std::cout << "Invalid input.\n";
        return 0;
    }

    // Validate common constraints
    if (salary < 10000 || salary > 1000000) {
        std::cout << "Invalid input.\n";
        return 0;
    }

    std::unique_ptr<Employee> emp;

    if (employeeType == 1) {
        // Manager
        int rating;
        if (!(std::cin >> rating)) {
            std::cout << "Invalid input.\n";
            return 0;
        }

        // Validate constraints
        if (rating < 1 || rating > 5) {
            std::cout << "Invalid input.\n";
            return 0;
        }

        emp = std::make_unique<Manager>(name, id, salary, rating);
    } else if (employeeType == 2) {
        // Developer

```



```

int extraHours;
if (!(std::cin >> extraHours)) {
    std::cout << "Invalid input.\n";
    return 0;
}

// Validate constraints
if (extraHours < 0 || extraHours > 100) {
    std::cout << "Invalid input.\n";
    return 0;
}

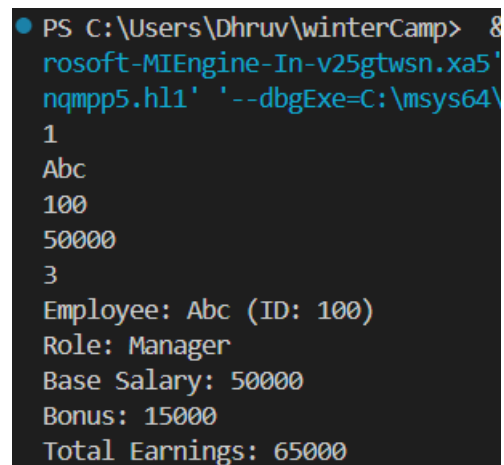
emp = std::make_unique<Developer>(name, id, salary, extraHours);
}

if (emp) {
    emp->displayEarnings();
} else {
    // This case should not be reached due to earlier checks,
    // but we include it for completeness.
    std::cout << "Invalid input.\n";
}

return 0;
}

```

Output:



```

PS C:\Users\Dhruv\winterCamp> 8
Microsoft-MIEngine-In-v25gtwsn.xa5'
nqmp5.hl1' '--dbgExe=C:\msys64\
1
Abc
100
50000
3
Employee: Abc (ID: 100)
Role: Manager
Base Salary: 50000
Bonus: 15000
Total Earnings: 65000

```