**Employee Compensation System**

**INTRODUCTION**

Employee Compensation System is a C++ console program developed based on the Object-Oriented Programming (OOP) principles. This system is designed to help manage various pay components of employees, such as base salary, allowances, and bonuses. By incorporating features like dynamic memory allocation, abstraction, and polymorphism, the system provides a flexible way to compute total salary, calculate commission-based incentives and view total compensation. Its interactive menu-driven interface allows users to perform operations like adding, removing, and displaying pay components in real time. This project serves as a practical demonstration of how core OOP concepts can be applied to solve real-world payroll management problems. Thus this document will demonstrate assigned task, explains how it was completed and includes annotated code with comments detailing the purpose of each line of the Employee Compensation system.

1. **Assigned Task**

The objective of this project is to design and implement an **Employee Compensation System** using **Object-Oriented Programming** principles in **C++**. The goal is to simulate a compensation model that allows management of various employee pay components in a modular and dynamic way.

**Functional Requirements:**

The system must provide the following features:

1. **Add Pay Component**: Allow users to add salary components (e.g: base salary, transport allowance, housing allowance).
2. **Remove Pay Component**: Enable removal of components by their index position.
3. **Display Components**: View all the added pay components with their descriptions and monetary amounts.
4. **Compute Total Salary**: Calculate the total of all pay components.
5. **Compute Commission**: Calculate a 10% commission based on each pay component.
6. **Compute Total Compensation**: Display the combined value of salary and commission.
7. **Exit**: Terminate the program.

**Learning Objective:**

The task demonstrates mastery of key OOP concepts including:

* **Abstraction**: via an abstract base class that defines a general calculator interface.
* **Polymorphism**: through inheritance and use of virtual functions for different compensation calculators.
* **Dynamic Memory Handling**: using pointers to dynamically manage pay component arrays.
* **Modular Design**: separating logic for input, display, calculation, and data management.
* **Pointer Arithmetic**: Accessing pay components using pointers instead of indexing to demonstrate low-level control.

1. **How Tasks It Was Completed**

The Employee Compensation System was developed using **C++**, applying Object-Oriented Programming principles to ensure modularity, flexibility, and clarity in design. The main building block of the program is a PayComponent structure which holds two key pieces of information for each salary entry, a character array for the description and a floating-point number representing the monetary value of that component. These components are stored in a dynamic array to allow flexible addition and removal at runtime.

At the heart of the system is an **abstract class** named CompCalculator, which defines a pure virtual function compute(). This function establishes a common interface for all types of compensation calculations. Two concrete classes inherit from this base: SalaryCalculator, which computes the total salary by summing all component amounts, and CommissionCalculator, which calculates a 10% commission on each component. These classes demonstrate **abstraction** and **polymorphism** by allowing the same compute() method to behave differently depending on the calculator used.

To manage user input and interaction, the program includes a **menu-driven loop** implemented in the main() function. Users can continuously perform actions such as adding new components, removing existing ones by index, displaying all entries, and performing various calculations. Each operation is encapsulated in its own function for clarity and reusability. For example, the addComponent() function expands the dynamic array to include a new component, while removeComponent() shrinks the array after deleting a specific item. The dynamic memory is carefully handled using new and delete[] to avoid memory leaks.

Throughout the system, key OOP concepts are applied to enhance design quality. **Abstraction** ensures calculators have a unified interface, **polymorphism** enables interchangeable use of different calculator types, and **modularity** keeps logic well-separated and reusable. This design approach not only fulfills the functional requirements but also makes the codebase scalable and easier to maintain or enhance in the future.

**3. Annotated Code with comments**

**#include <iostream>** //Includes the standard input/output stream library. Needed to use cout, cin, etc.

**#include <cstring>** // Includes string handling (for char arrays)

**using namespace std;** // Enables use of standard names (cout, cin) without prefixing 'std::'

**struct PayComponent** { // Define a struct to hold employee pay details

char desc[30]; // Description of the pay component (e.g., 'Base Salary')

float amount; // Amount of the component in currency

};

**class CompCalculator** { //Abstract base class for salary-related calculations

public:

virtual float compute(const PayComponent\* comps, int n) = 0; // Pure virtual method for calculation

virtual ~CompCalculator() {} // Virtual destructor for polymorphic deletion

};

**class SalaryCalculator : public CompCalculator** { //Create Derived class to calculate total salary

public:

float compute(const PayComponent\* comps, int n) override {

float total = 0; // Initialize total

for (int i = 0; i < n; ++i) {

total += comps[i].amount; // Add each component's amount to total

}

return total; // Return total salary

}

};

**class CommissionCalculator : public CompCalculator {** //Create Derived class to calculate commission from all components

public:

float compute(const PayComponent\* comps, int n) override {

float commission = 0; // Initialize commission

for (int i = 0; i < n; ++i) {

commission += comps[i].amount \* 0.10f; // Add 10% of each component's amount

}

return commission; // Return total commission

}

};

// Function to add a new pay component to the array

**void addComponent(PayComponent\*& comps, int& n, const PayComponent& newComp)**

{

PayComponent\* temp = new PayComponent[n + 1]; // Create new larger array

for (int i = 0; i < n; ++i) {

temp[i] = comps[i]; // Copy old data to new array

}

temp[n] = newComp; // Add new component at the end

delete[] comps; // Free or delete old memory

comps = temp; // Point components to new array

++n; //this line is an Increase count

}

// Function to remove a component by index

**void removeComponent(PayComponent\*& comps, int& n, int index)**

{

if (index < 0 || index >= n) return; // Validate index

PayComponent\* temp = new PayComponent[n - 1]; // Create smaller array

for (int i = 0, j = 0; i < n; ++i) {

if (i != index) temp[j++] = comps[i]; // Copy all but the one to remove

}

delete[] comps; // Free or delete old memory

comps = temp; // Point to new array

--n; // Decrease count

}

// Function to display all pay components

**void showComponents(const PayComponent\* comps, int n)**

{

if (n == 0) {

cout << "No components found.\n"; // Show message if list is empty

return;

}

for (int i = 0; i < n; ++i) {

cout << i << ". " << comps[i].desc // Display index and description

<< " - $" << comps[i].amount << endl; // Display amount

}

}

// Main function: user interaction loop

**int main()** {

int numComponents = 0; // Initialize number of components

PayComponent\* comps = nullptr; // Pointer to dynamic array of components

CompCalculator\* salaryCalc = new SalaryCalculator(); //Create calculator objects for salary and commission

CompCalculator\* commissionCalc = new CommissionCalculator();

int choice; // To store user's menu selection

do {

//using do while loop to display user choice

**// Display menu options**

cout << "\n--- Employee Compensation System ---\n";

cout << "1. Add a pay component\n";

cout << "2. Remove a component by index\n";

cout << "3. Show all components\n";

cout << "4. Compute total salary\n";

cout << "5. Compute total commission\n";

cout << "6. Compute total compensation (salary + commission)\n";

cout << "0. Exit\n";

cout << "Choose an option: "; // Message to ask user to choose an option

cin >> choice; // allow user to input choice

switch (choice) {

case 1: {

PayComponent newComp;

cout << "Enter description: ";

cin.ignore(); // Clear input buffer

cin.getline(newComp.desc, 30); // Read description

cout << "Enter amount: ";

cin >> newComp.amount; // Read amount

addComponent(comps, numComponents, newComp); // Add to list

cout << "Component added!\n";

break;

}

case 2: {

showComponents(comps, numComponents); // Show list

int index;

cout << "Enter index to remove: ";

cin >> index; // Get index

removeComponent(comps, numComponents, index); // Remove item

cout << "Component removed.\n";

break;

}

case 3:

showComponents(comps, numComponents); // Show all items

break;

case 4:

cout << "Total Salary: $" // Display total salary

<< salaryCalc->compute(comps, numComponents)

<< endl;

break;

case 5:

cout << "Total Commission (10%): $" // Display total commission from all components

<< commissionCalc->compute(comps, numComponent << endl;

break;

case 6: {

float sal = salaryCalc->compute(comps, numComponents); // Calculate salary

float com = commissionCalc->compute(comps, numComponents); // Calculate commission

cout << "Total Salary: $" << sal << endl;

cout << "Total Commission: $" << com << endl;

cout << "Total Compensation: $" << sal + com << endl; // Display total Compensation

break;

}

case 0:

cout << "Exiting...\n"; // Exit message

break;

default:

cout << "Invalid option. Try again.\n"; // Error handling

}

} while (choice != 0); // Repeat until user exits

**// Free dynamically allocated memory**

delete[] comps;

delete salaryCalc;

delete commissionCalc;

return 0; // Exit program

}