|  |  |
| --- | --- |
|  | **Practical** |

|  |  |
| --- | --- |
| **Program:3.1** | .A growing organization sought to enhance its payroll process to improve accuracy and efficiency in calculating employee salaries. The goal was to manage details such as the employee's name, basic salary, and bonus amount. While most employees would receive a default bonus, there was a need for flexibility to provide customized bonuses for exceptional performers or those in specialized roles.  To ensure real-time computation of each employee's total salary, the system would dynamically calculate the total salary by adding the basic salary and the bonus. An inline function was chosen for this task, allowing the system to compute the total salary instantly during processing. The system was designed to initialize employee records using a constructor, where the bonus could either be set to a default value or adjusted according to specific employee criteria.  As the organization expanded, managing employee records efficiently became a key consideration. The solution involved storing employee details dynamically, ensuring the system could handle a growing number of records while maintaining scalability and flexibility. Each employee record would include their personal details, basic salary, and bonus information, with the system displaying a breakdown of each employee’s details along with their total salary. |
| **Code** | #include <iostream>  #include <string> using namespace std;    class Employee { private:  string name; |

|  |  |
| --- | --- |
|  | float base; float bonus;    public:  // Constructor with default bonus  Employee(string n, float b, float bn = 1000.0) { name = n; base = b; bonus = bn;  }    // Inline function to calculate total salary inline float getSalary() { return base + bonus;  }    // Display Employee Details void showDetails() { cout << "\nEmployee Name: " << name << endl; cout << "Basic Salary: $" << base << endl; cout << "Bonus: $" << bonus << endl; cout << "Total Salary: $" << getSalary() << endl;  }  };    int main() { int n;  cout << "Enter number of employees: "; cin >> n; cin.ignore(); // Ignore trailing newline    Employee \*emp[100]; // Array of employee pointers    // Create Multiple Employee Records  for (int i = 0; i < n; i++) { string name; float b, bn; int choice;    cout << "\nEnter details for employee " << i + 1 << ":\n"; cout << "Employee Name: "; getline(cin, name); cout << "Basic Salary: "; cin >> b; |

|  |  |
| --- | --- |
|  | cout << "Do you want to enter a custom bonus? (1-Yes, 2-No): "; cin >> choice;    if (choice == 1) { cout << "Enter Bonus: "; cin >> bn; emp[i] = new Employee(name, b, bn);  } else {  emp[i] = new Employee(name, b);  } cin.ignore();  }    cout << "\nDisplaying Employee Details:\n"; for (int i = 0; i < n; i++) { emp[i]->showDetails(); cout << "--------------------------\n"; }  for (int i = 0; i < n; i++) { delete emp[i];  }    cout << "Name: preet " << endl  << "ID: 24ce091" << endl; return 0;  } |
| **Output** |  |

|  |  |
| --- | --- |
|  | **Practical** |
| **Program:3.2** | A software development company was tasked with conducting a performance analysis of recursive algorithms versus their iterative  counterparts. The specific focus was on calculating the sum of integers in an array, where the array's size and elements were to be provided dynamically by the user. To facilitate memory management and enable dynamic resizing of the array, the team decided to use a flexible container for storing the array elements.  The system was designed to first prompt users for the array's size and then request the input of individual elements. A recursive function was to be implemented to compute the sum by dividing the problem into smaller sub-problems, recursively summing subsets of the array until reaching the base  case. In addition to the recursive implementation, an iterative version of the function would be created for comparison.  The main objective of the study was to assess and compare the  computational performance and implementation complexity of both recursive and non-recursive approaches. By evaluating execution times, memory usage, and code complexity, the team hoped to gain insights into the trade-offs between recursion and iteration, particularly in terms of efficiency and applicability to real-world problems. |
| **Code** |  |

|  |  |
| --- | --- |
|  | **#include <iostream>**  **#include <vector>**  **using namespace std;**  **int recSum(vector<int> &arr, int n) {**  **if (n == 0)**  **return 0;**  **return arr[n - 1] + recSum(arr, n - 1);**  **}**  **int iterSum(vector<int> &arr) {**  **int sum = 0;**  **for (int i = 0; i < arr.size(); i++)**  **sum += arr[i];**  **return sum;**  **}**  **int main() {**  **int n;**  **cout << "Enter size of array: ";**  **cin >> n;**  **vector<int> arr(n);**  **cout << "Enter " << n << " elements:\n";**  **for (int i = 0; i < n; i++)**  **cin >> arr[i];**  **int sumRec = recSum(arr, n);**  **int sumIter = iterSum(arr);**  **cout << "\nRecursive Sum: " << sumRec << endl;**  **cout << "Iterative Sum: " << sumIter << endl;**  **cout << "Name: PREET" << endl**  **<< "ID: 24CE091" << endl;**  **return 0;** |
| **Output** |  |

|  |  |
| --- | --- |
|  | **Practical** |
| **Program:3.3** | A community bank sought to enhance its account management system with a digital solution to improve efficiency and provide better customer service. The system was required to manage the essential details of each account, including the account number, account holder’s name, and balance.  Additionally, the bank wanted to provide a secure mechanism for transferring money between accounts, allowing customers to easily manage their funds.  The bank also needed a way to track the total number of accounts created, which would be important for generating reports and understanding the growth of their customer base. This feature was aimed at helping the bank maintain an overview of their account portfolio and analyze trends over time.  To ensure smooth and reliable operations, the system was designed to store account information in a way that would allow easy access and updates. When new accounts were created, they would be added to the system dynamically. The management team planned for future scalability and performance improvements by considering more efficient storage and retrieval methods after the initial system was built, ensuring that the bank could easily accommodate more accounts and transactions as the customer base grew. |
| **Code** |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
| **Output** |  |

|  |  |
| --- | --- |
|  | **Practical** |
| **Program:3.4** | A technology firm aimed to develop a flexible and reusable solution for managing collections of various data types, including integers, floating-point numbers, and characters. The system was intended to perform fundamental operations on these collections, such as finding the maximum value, reversing the collection, and displaying all elements. To achieve versatility and avoid redundancy in code, the solution was designed to use function templates, allowing the same logic to be applied seamlessly to different data types.  The team recognized the importance of using dynamic arrays to store the collections, enabling efficient management of varying collection sizes. The design emphasized scalability and flexibility, ensuring that the system could handle different data types and their associated operations with minimal changes to the core logic.  In practice, the system allowed for the creation of collections for various data types, such as integers, floating-point numbers, and characters. The operations on these collections included determining the maximum value, reversing the order of elements, and printing the collection contents |
| **Code** | #include <iostream> #include <vector> using namespace std;    template <typename T> void display(vector<T> &arr) {  for (int i = 0; i < arr.size(); i++) cout << arr[i] << " "; cout << endl;  }    template <typename T>  T getMax(vector<T> &arr) { T maxVal = arr[0]; for (int i = 1; i < arr.size(); i++) { if (arr[i] > maxVal) |

|  |  |
| --- | --- |
|  | maxVal = arr[i];  }  return maxVal;  }    template <typename T> void reverse(vector<T> &arr) { int n = arr.size(); for (int i = 0; i < n / 2; i++) swap(arr[i], arr[n - i - 1]);  }    int main() { vector<int> intArr = {5, 8, 1, 3, 9}; vector<float> floatArr = {2.3, 5.5, 1.1, 4.8}; vector<char> charArr = {'a', 'z', 'm', 'b'};    cout << "Original Integer Array: "; display(intArr);  cout << "Max Integer: " << getMax(intArr) << endl; reverse(intArr);  cout << "Reversed Integer Array: "; display(intArr);    cout << "\nOriginal Float Array: "; display(floatArr); cout << "Max Float: " << getMax(floatArr) << endl; reverse(floatArr); cout << "Reversed Float Array: "; display(floatArr);    cout << "\nOriginal Character Array: "; display(charArr); cout << "Max Character: " << getMax(charArr) << endl; reverse(charArr); cout << "Reversed Character Array: "; display(charArr);    cout << "\nName: PREET" << endl  << "ID: 24CE091" << endl;    return 0;  } |
| **Output** |  |

|  |  |
| --- | --- |
|  | **Practical** |
| **Program:3.5** | A data analytics company was tasked with developing a unique digital signature system based on the concept of "super digits." The system required finding a single-digit representation of a given number through recursive digit summation. The algorithm was defined as follows:  If the number has only one digit, it is its super digit. Otherwise, the super digit is the super digit of the sum of its digits, repeated recursively until a single digit is obtained.  The challenge involved an additional complexity—constructing the number by concatenating a given string representation of an integer multiple times. For example, if the number n was represented as a string and concatenated k times, the super digit of the resulting number needed to be calculated.  For instance:  Given n = 9875 and k = 4, the number is represented as 9875987598759875.  The sum of digits in this number is calculated recursively until a single digit remains: 9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 + 9 + 8 + 7 + 5 = 116 1 + 1 + 6 = 8  The super digit is 8.  The system was required to handle large numbers efficiently by leveraging mathematical insights rather than explicitly constructing large concatenated strings. This case study called for implementing a recursive solution to calculate the super digit, supported by a mathematical approach to optimize the handling of repeated sums. |

|  |  |
| --- | --- |
| **Code** | #include <iostream> using namespace std;    int getSum(long long n) { if (n < 10) return n; long long sum = 0; while (n > 0) { sum += n % 10; n /= 10;  }  return getSum(sum);  }    int superDigit(string n, int k) { long long sum = 0; for (char c : n) sum += (c - '0');    sum \*= k;  return getSum(sum);  }    int main() { string n; int k;  cout << "Enter number (as string): "; cin >> n; cout << "Enter repetition count: "; cin >> k;    int result = superDigit(n, k); cout << "Super Digit: " << result << endl;    cout << "\nName: PREET << endl  << "ID: 24CE091" << endl;    return 0;  } |
| **Output** |  |