1. Write a program to find the sum for the given variables using function with default arguments.

AIM:

To write a program that finds the sum of two or three variables using a function that has default arguments for the third variable.

DEFINITION:

Default arguments are arguments that have a predefined value in the function declaration. They are used when the function is called without passing the actual arguments for those parameters. This way, we can use the same function for different numbers of arguments without overloading it.

SYNTAX:

```
return_type add (int x, int y, int z = 0);
{
//can add 2 or 3 variables based on function call
}
```

PROCEDURE:

The add function calculates the sum of two or three numbers. It takes two mandatory arguments, x and y, and an optional third argument, z, which is assumed to be 0 if not provided.
In the main function:
Two integer variables, firstNum and secondNum, are declared and initialized with values 10 and 20 respectively.
Another integer variable, thirdNum, is declared and set to 30.
The add function is called twice in the main function:
The first call uses firstNum and secondNum as arguments and prints their sum.
The second call uses all three variables and displays the sum of the three numbers.
cout is used to display the results of the function calls along with appropriate messages.
The program ends by returning 0 from the main function.

SOURCE CODE:

```
#include <iostream>
using namespace std;
// A function that adds two or three numbers and gives the total
// The third number is optional and assumed to be 0 if not given
int add(int x, int y, int z = 0) {
  // Calculate and return the sum of x, y, and z
  return x + y + z;
}

int main() {
  // Declare and set the values of two variables
  int firstNum = 10, secondNum = 20;
  // Declare and set the value of another variable
  int thirdNum = 30;
  // Call the "add" function with two numbers, and display the result
  cout << "The sum of " << firstNum << " and " << secondNum << " is " << add(firstNum,
secondNum) << endl;
  // Call the "add" function with three numbers, and show the result
  cout << "The sum of " << firstNum << ", " << secondNum << "and " << thirdNum << " is " <<
add(firstNum, secondNum, thirdNum) << endl;
  return 0;
}
```

OUTPUT:

Correct Output:

```
The sum of 10 and 20 is 30
The sum of 10,20 and 30 is 60
```

Wrong Output:

```
The sum of 10 and 20 is 60
The sum of 10,20 and 30 is 20
```

Error Output:

```
The sum of 10 and 0 is 30
The sum of 10, 30 and 30 is 0
```

2. Write a program to swap the values of two variables and demonstrates a function using call by value.

AIM:

To write a program that swaps the values of two variables using a function that takes the arguments by value and not by reference.

DEFINITION:

Call by value is a parameter-passing mechanism used in programming languages. When a function is called with arguments, the values of the arguments are copied into the function's parameters. This means that the function works with copies of the original values, and any modifications made to the parameters within the function do not affect the original arguments in the calling code.

SYNTAX:

```
return_type swapValues(int a, int b) {

   int temp = a;

   a = b;

   b = temp;

}
```

PROCEDURE:

Function Definition (`swapValues`): Define a function named `swapValues` that takes two integer parameters, `a` and `b`.
Inside the function, swap the values of `a` and `b` using a temporary variable, but note that this doesn't impact the original variables.
Main Function: Declare and initialize two integer variables, `x` and `y`.
Display Initial Values: Display the initial values of `x` and `y` using `cout`.
Function Call (`swapValues`): Call the `swapValues` function with the arguments `x` and `y`.
Display Values After Swapping: Display the values of `x` and `y` again using `cout`.
Program Output: Observe the output, noting that the swapping inside the function does not affect the original `x` and `y` values outside the function.
Program Termination: The program ends, returning 0 from the `main` function.

SOURCE CODE:

```
#include <iostream>

using namespace std;

// Function to swap the values of two variables by value

void swapValues(int a, int b) {

   // Create a temporary variable to hold the value of 'a'

   int temp = a;

   // Assign the value of 'b' to 'a'

   a = b;

   // Assign the temporary value to 'b', effectively swapping the values

   b = temp;

   cout << "Inside swapping function : x = " << a<< ", y = " << b << endl;

}

int main() {

   // Declare and initialize two integer variables

   int x = 5, y = 10;

   // Display the values before swapping

   cout << "Before swapping: x = " << x << ", y = " << y << endl;

   // Call the swapValues function, but it won't affect the original variables

   swapValues(x, y);

   // Display the values after swapping, which remain unchanged

   cout << "After swapping: x = " << x << ", y = " << y << endl;

   return 0;

}
```

OUTPUT:-

Correct Output:

```
Before swapping: a = 10, b = 20
Inside swapping function: x = 20, y = 10
After swapping: a = 10, b = 20
```

Wrong Output:

```
Before swapping: a = 10, b = 20
Inside swapping function: x = 20, y = 10
After swapping: a = 20, b = 10
```

Error Output:

```
 Before swapping: a = 10, b = 20
Inside swapping function: x = 90, y = 0
After swapping: a = 0, b = 20
```

3. Write a program to create a template function for Bubble Sort and demonstrate sorting of integers and doubles.

AIM :-

To Write a user driven program to create a template function for Bubble Sort and demonstrate sorting of integers and doubles.

DEFINITION: -

A template in programming is a mechanism that allows you to create a blueprint for a generic function or class. Instead of specifying a specific data type or structure, you can define a template that can work with various data types or structures. Templates are used to create reusable and flexible code that can adapt to different data types without duplicating code.

SYNTAX: -

```
template <typename T>

return_type function_name(T parameter1, T parameter2, ...) {

   // Function implementation

}
```

PROCEDURE:-

**Implement a template function named bubbleSort that takes an array and its size.
Use the bubble sort algorithm to sort the array elements.
In the main function, prompt the user to choose between integers and doubles.
Based on the user's choice:
Declare an array of the chosen data type and input its elements.
Call bubbleSort to sort the array.
Display the sorted array.**

SOURCE Code:

```
#include <iostream>

using namespace std;

// Template function for Bubble Sort

template <typename T>

void bubbleSort(T arr[], int size) {                    //template for bubble sort

   for (int i = 0; i < size - 1; ++i) {

      for (int j = 0; j < size - i - 1; ++j) {
           if (arr[j] > arr[j + 1]) {
               // Swap the elements
               T temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
           }
      }
   }
}
int main() {                        //driving portion
   int choice;
   cout << "Choose sorting type:" << endl;
   cout << "1. Integers" << endl;//asking user input
   cout << "2. Doubles" << endl;
   cout << "Enter your choice: ";
   cin >> choice;
   switch (choice) {
      case 1: {                        //for integers
           int intSize;
           cout << "Enter the number of integers: ";
           cin >> intSize;
           int intArr[intSize];
           cout << "Enter " << intSize << " integers: ";
           for (int i = 0; i < intSize; ++i) {
               cin >> intArr[i];
           }
           bubbleSort(intArr, intSize);
           cout << "Sorted integers: ";        //print sorted integers
           for (int i = 0; i < intSize; ++i) {
               cout << intArr[i] << " ";
           }
           cout << endl;
           break;
      }
      case 2: {                     //for doubles
           int doubleSize;
           cout << "Enter the number of doubles: ";
           cin >> doubleSize;
           double doubleArr[doubleSize];
           cout << "Enter " << doubleSize << " doubles: ";
           for (int i = 0; i < doubleSize; ++i) {
               cin >> doubleArr[i];
           }
           bubbleSort(doubleArr, doubleSize);
           cout << "Sorted doubles: ";             //print sorted doubles
           for (int i = 0; i < doubleSize; ++i) {
               cout << doubleArr[i] << " ";
           }
           cout << endl;
           break;
      }
      default:
           cout << "Invalid choice." << endl;    //wrong choice
           break;
           }
           return 0;
      }
```

<u>OUTPUT:-</u>

Correct Output:

```
Choose sorting type:
1. Integers
2. Doubles
Enter your choice: 2
Enter the number of doubles: 4
Enter 4 doubles: 2.5 1.1 3.7 2.0
Sorted doubles: 1.1 2 2.5 3.7
```

Wrong Output:

```
Choose sorting type:
1. Integers
2. Doubles
Enter your choice: 3
Invalid choice.
```

Error Output:

```
 Choose sorting type:
1. Integers
2. Doubles
Enter your choice: 1
Enter the number of integers: -3
Enter -3 integers: [Program crashes due to invalid input]
```

4. Define a STUDENT class with USN, Name, and Marks in 3 tests of a subject. Declare an array of 10 STUDENT objects. Using appropriate functions, find the average of the two better marks for each student. Print the USN, Name and the average marks of all the students.

AIM :-

To create a STUDENT class with USN, Name, and Marks in 3 tests of a subject. Declare an array of 10 STUDENT objects. Using appropriate functions, find the average of the two better marks for each student. Print the USN, Name and the average marks of all the students.

DEFINITION:-

It defines a class named STUDENT that represents information about students and their test scores. The program collects details for 10 students, calculates and displays the average marks for each student based on their three test scores, and then prints out the student's name, USN (University Serial Number), and average marks.

SYNTAX:-

```
class STUDENT {

private:

   string USN;

   string Name;

   int Marks[3];

public:

   void input();

   float calculateAverage();

   void display();

};
```

PROCEDURE:

We define a class named STUDENT with private members for USN, Name, and Marks.

The input() function is used to input the details (USN, Name, and Marks) for a student.

The calculateAverage() function calculates the average of the two best marks out of three for a student.

The display() function displays the details of a student along with their average marks.

Inside the main() function:

We declare an array of 10 STUDENT objects.

We use a loop to input details for each student.

We use another loop to display details and average marks for each student.

This program lets you input details for 10 students, calculates the average of their two best marks, and displays the details along with the calculated average marks for each student.

SOURCE Code:

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

// Define the STUDENT class
class STUDENT {
private:
   string USN;
   string Name;
   int Marks[3];
public:
   // Function to input student details
   void input() {
      cout << "Enter USN: ";
      cin >> USN;
      cout << "Enter Name: ";
      cin.ignore(); // Clear the newline character from the buffer
      getline(cin, Name);
      cout << "Enter Marks in 3 tests: ";
      for (int i = 0; i < 3; ++i) {
         cin >> Marks[i];
      }
   }
```

```
    // Function to calculate the average of the two best marks
    float calculateAverage() {
        // Find the two highest marks among three
        int max1 = Marks[0], max2 = Marks[1];
        for (int i = 2; i < 3; ++i) {
            if (Marks[i] > max1) {
                max2 = max1;
                max1 = Marks[i];
            } else if (Marks[i] > max2) {
                max2 = Marks[i];
            }
        }
        // Calculate and return the average of two best marks
        return static_cast<float>(max1 + max2) / 2;
    }
    // Function to display student details along with average marks
    void display() {
        cout << "USN: " << USN << ", Name: " << Name << ", Average Marks: " << calculateAverage()
<< endl;
    }
};
int main() {
    STUDENT students[10]; // Array of 10 STUDENT objects

    // Input details for each student
    for (int i = 0; i < 10; ++i) {
        cout << "Enter details for student " << i + 1 << ":" << endl;
        students[i].input();
    }
    // Display details for each student along with average marks
    for (int i = 0; i < 10; ++i) {
        cout << "Details for student " << i + 1 << ":" << endl;
        students[i].display();
    }
    return 0;
}
```

OUTPUT:-

Correct Output:

```
Enter details for student 1:
Enter USN: 12345
Enter Name: John Doe
Enter Marks in 3 tests: 85 92 78
Enter details for student 2:
Enter USN: 67890
Enter Name: Jane Smith
Enter Marks in 3 tests: 76 89 95
...
Enter details for student 10:
Enter USN: 54321
Enter Name: Alice Johnson
Enter Marks in 3 tests: 82 75 88

Details for student 1:
USN: 12345, Name: John Doe, Average Marks: 86.5
Details for student 2:
USN: 67890, Name: Jane Smith, Average Marks: 92
...
Details for student 10:
USN: 54321, Name: Alice Johnson, Average Marks: 85
```

Wrong Output:

```
Enter details for student 1:
Enter USN: 12345
Enter Name: John Doe
Enter Marks in 3 tests: 85 92 78
Enter details for student 2:
Enter USN: 67890
Enter Name: Jane Smith
Enter Marks in 3 tests: 76 89 95
...
Enter details for student 10:
Enter USN: 54321
Enter Name: Alice Johnson
Enter Marks in 3 tests: 82 75 88

Details for student 1:
USN: 12345, Name: John Doe, Average Marks: 86.5
Details for student 2:
USN: 67890, Name: Jane Smith, Average Marks: 92
...
Details for student 10:
USN: 54321, Name: Alice Johnson, Average Marks: 85
```

Error Output:

```
Enter details for student 1:
Enter USN: 12345
Enter Name: John Doe
Enter Marks in 3 tests: -85 92 78
[Program crashes due to invalid input]
```

5. Write a C++ program to create a class called COMPLEX and implement the following overloading functions ADD that return a complex number: (i) ADD (a, s2) – where 'a' is an integer (real part) and s2 is a complex number (ii) ADD (s1, s2) – where s1 and s2 are complex numbers.

AIM :-

To create a C++ program to create a class called COMPLEX and implement the following overloading functions ADD that return a complex number:

- **(i) ADD (a, s2) – where 'a' is an integer (real part) and s2 is a complex number**
- **(ii) (ii) ADD (s1, s2) – where s1 and s2 are complex numbers.**

DEFINITION :-

It defines a class named COMPLEX that represents complex numbers and demonstrates operator overloading. The class has two private data members: real for the real part and imag for the imaginary part of the complex number. It also provides constructors to create complex numbers, overloaded ADD functions to perform addition, and a display function to print the complex number.

SYNTAX :-

```
class Complex {

private:

   double real,imag;

public:

   Complex(double r, double i) : real(r), imag(i) {}

   // Overloading the + operator

   Complex operator+(const Complex& other) {

      return Complex(real + other.real, imag + other.imag);

   }};
```

PROCEDURE :-

Define COMPLEX Class: Define a class named COMPLEX with private data members for real and imaginary parts.
Include default and parameterized constructors to initialize the complex numbers.

Implement overloaded ADD functions to add an integer and complex numbers.
Implement a display function to show the complex number.

Inside the main function: Create two complex numbers c1 and c2.
Use the overloaded ADD function to add an integer and complex numbers, as well as two complex numbers.
Display the results using the display function.

<u>SOURCE CODE :-</u>

```
#include <iostream>

using namespace std;

class COMPLEX {

private:

   double real;

   double imag;

public:

   // Default constructor

   COMPLEX() : real(0), imag(0) {}

   // Parameterized constructor

   COMPLEX(double r, double i) : real(r), imag(i) {}

   // Overloaded ADD function to add an integer and a complex number

   COMPLEX ADD(int a, COMPLEX s2) {

      COMPLEX result;

      result.real = a + s2.real; // Add the integer to the real part of s2

      result.imag = s2.imag;     // Keep the imaginary part of s2 unchanged

      return result;

   }

   // Overloaded ADD function to add two complex numbers

   COMPLEX ADD(COMPLEX s1, COMPLEX s2) {

      COMPLEX result;

      result.real = s1.real + s2.real; // Add the real parts of s1 and s2
```

```
        result.imag = s1.imag + s2.imag; // Add the imaginary parts of s1 and s2

        return result;

    }

    // Display the complex number

    void display() {

        cout << real << " + " << imag << "i" << endl;

    }};

int main() {

    COMPLEX c1(2, 3); // Create a complex number c1 with real part 2 and imaginary part 3

    COMPLEX c2(4, 5); // Create a complex number c2 with real part 4 and imaginary part 5

    COMPLEX c3 = c1.ADD(10, c2); // Add 10 to the real part of c2 and keep its imaginary part
unchanged

    COMPLEX c4 = c1.ADD(c2, c3); // Add c2 and c3 using the overloaded ADD function for complex
numbers

    cout << "c3: ";

    c3.display(); // Display the value of c3

    cout << "c4: ";

    c4.display(); // Display the value of c4

    return 0;

}
```

OUTPUT :-

Correct Output:

```
c3: 14 + 5i
c4: 20 + 13i
```

Wrong Output:

```
c3: 15 + 5i
c4: 20 + 13i
```

Error Output:

[Program crashes due to division by zero]

6. Friend functions and friend classes:

a) Write a program to define class name HUSBAND and WIFE that holds the income respectively. Calculate and display the total income of a family using Friend function.

b) Write a program to accept the student detail such as name and 3 different marks by get_data() method and display the name and average of marks using display() method. Define a friend class for calculating the average of marks using the method mark_avg()

AIM :-

a) To Create a program to define class name HUSBAND and WIFE that holds the income respectively. Calculate and display the total income of a family using Friend function.

DEFINITION :-

A friend function in C++ is a function that is not a member of a class but is granted access to the private and protected members of that class. It can be invoked from outside the class just like any other function, but it has the privilege to access private and protected data members of the class it is declared as a friend to. Friend functions are typically used when you need external functions to work closely with class internals without violating encapsulation.

SYNTAX:-

```
class MyClass {
private:
   // Private data members
public:
   // Public member functions
   friend returnType friendFunctionName(parameters); // Friend function declaration
};
returnType friendFunctionName(parameters) {
   // Implementation of the friend function
}
```

PROCEDURE:-

The program defines two classes HUSBAND and WIFE, each with a private member income. A friend function calculateTotalIncome is declared within both classes. This function allows

accessing the private members of both classes to calculate the total income.
In the main function, the user is prompted to enter the incomes of the husband and wife.
Objects of the HUSBAND and WIFE classes are created with the provided incomes.
The calculateTotalIncome friend function is then used to calculate the total family income.
The calculated total income is displayed to the user.

SOURCE CODE:-

```
#include <iostream>

using namespace std;

// Forward declaration of the WIFE class

class WIFE;

// Class to represent the HUSBAND

class HUSBAND {

private:

   double incomeHusband; // Private data member to store the husband's income

public:

   // Constructor to initialize the husband's income

   HUSBAND(double income) : incomeHusband(income) {}

   // Declare the friend function that calculates total income

   friend double calculateTotalIncome(HUSBAND husband, WIFE wife);

}

// Class to represent the WIFE

class WIFE {

private:

   double incomeWife; // Private data member to store the wife's income


public:

   // Constructor to initialize the wife's income

   WIFE(double income) : incomeWife(income) {}

   // Declare the friend function that calculates total income
```

```
    friend double calculateTotalIncome(HUSBAND husband, WIFE wife);
};
// Friend function to calculate the total family income
double calculateTotalIncome(HUSBAND husband, WIFE wife) {
    double totalIncome = husband.incomeHusband + wife.incomeWife;
    return totalIncome;
}
int main() {
    double husbandIncome, wifeIncome;
    // Input husband's income
    cout << "Enter husband's income: ";
    cin >> husbandIncome;
    // Input wife's income
    cout << "Enter wife's income: ";
    cin >> wifeIncome;
    // Create objects for husband and wife
    HUSBAND husband(husbandIncome);
    WIFE wife(wifeIncome);
    // Calculate total family income using the friend function
    double totalIncome = calculateTotalIncome(husband, wife);
    // Display the total family income
    cout << "Total family income: $" << totalIncome << endl;
    return 0;
}
```

OUTPUT:-

Correct Output:

```
Enter husband's income: 50000
Enter wife's income: 40000
Total family income: $90000
```

Wrong Output:

```
Enter husband's income: 50000
Enter wife's income: 40000
Total family income: $95000
```

Error Output:

```
Enter husband's income: abc
Enter wife's income: 40000
[Program crashes due to invalid input]
```

6B)

AIM:-

b) Write a program to accept the student detail such as name and 3 different marks by get_data() method and display the name and average of marks using display() method. Define a friend class for calculating the average of marks using the method mark_avg()

DEFINITION:-

It demonstrates the use of friend classes and static member functions. The program defines two classes, Student and MarkCalculator, which interact to calculate and display the average marks of a student.

SYNTAX:-

```
class FriendClass;
class MainClass {
   friend class FriendClass; // Declare FriendClass as a friend
   // MainClass members
};
class FriendClass {
   // FriendClass members
};
class StaticExample {
public:
   static void staticFunction(int value) {
      // Implementation of the static function
   }
};
int main() {
   MainClass mainObject;
   FriendClass friendObject;
   // Calling the static function using the class name
   StaticExample::staticFunction(5);
   return 0;
}
```

PROCEDURE:-

Define a Student class with private data members for the student's name and an array to store three marks.
Implement member functions get_data() to input student data and display() to display the student's name.
Declare the MarkCalculator class as a friend of the Student class, allowing it to access private members.
Define a MarkCalculator class.
Declare a static member function mark_avg(const Student& student) to calculate the average of three marks using a reference to a Student object.
Inside the main function:
Create a Student object.
Get student data using get_data() and display the name using display().
Calculate the average marks using MarkCalculator::mark_avg(student).
Display the calculated average marks.

SOURCE CODE:-

```
#include <iostream>
using namespace std;
class MarkCalculator; // Forward declaration
class Student
{
private:
string name;
float marks[3]; // Array to store three marks
public:
void get_data()
{
cout << "Enter student name: ";
cin >> name;
cout << "Enter three marks: ";
for (int i = 0; i < 3; i++)
{
cin >> marks[i]; // Inputting three marks
}
}
void display()
{
cout << "Name: " << name << endl; // Displaying student's name
}
friend class MarkCalculator; // Allowing MarkCalculator class to access private members
};
class MarkCalculator
{
public:
static float mark_avg(const Student& student)
{
float sum = 0;
for (int i = 0; i < 3; i++)
{
sum += student.marks[i]; // Calculating the sum of marks
}
return sum / 3; // Calculating and returning the average of marks
}
};

int main()
{
Student student;
student.get_data(); // Getting student data (name and marks)
student.display(); // Displaying student's name
float average = MarkCalculator::mark_avg(student); // Calculating average marks
cout << "Average marks: " << average << endl; // Displaying the average marks
return 0;
}
```

OUTPUT:-

Correct Output:

```
Enter student name: Alice
Enter three marks: 85 90 88
Name: Alice
Average marks: 87.6667
```

Wrong Output:

```
Enter student name: Bob
Enter three marks: 75 80 82
Name: Bob
Average marks: 85
```

Error Output:

```
Enter student name: Carol
Enter three marks: 92 xyz 88
Name: Carol
Average marks: NaN
```

**7. Create a class called MATRIX using two-dimensional array of integers. Implement the following operations by overloading the operator == which checks the compatibility of two matrices to be added and subtracted. Perform the addition and subtraction by overloading the + and – operators respectively. Display the results by overloading the operator <<. If (m1== m2) then
m3 = m1+m2 and m4 = m1- m2 else display error.**

AIM :-

**To Create a class called MATRIX using two-dimensional array of integers. Implement the following operations by overloading the operator == which checks the compatibility of two matrices to be added and subtracted. Perform the addition and subtraction by overloading the + and – operators respectively. Display the results by overloading the operator <<.
If (m1== m2) then
m3 = m1+m2 and m4 = m1- m2 else display error.**

DEFINITION :-

It demonstrates the usage of operator overloading and dynamic memory allocation within a class for creating and manipulating matrices. The program defines a MATRIX class that allows you to perform addition and subtraction of matrices and compares them for compatibility.

SYNTAX:-

```
class MATRIX {
   // Class members and methods
public:
   MATRIX operator+(const MATRIX& other) const {
      MATRIX result(row, col);
      // Perform matrix addition here
      return result;
   }
};
```

PROCEDURE:-

Define a class MATRIX with private data members for rows, columns, and a 2D dynamic array.Implement a constructor to allocate memory for the matrix based on rows and columns.Implement a member function matrix_element() to input matrix elements.Overload operators ==, +, -, and << for matrix comparison, addition, subtraction, and output.

In the main() function:Input the number of rows and columns for the first matrix.Create an instance m1 of the MATRIX class.Input matrix elements for m1.Input the number of rows and columns for the second matrix.Create an instance m2 of the MATRIX class.Input matrix elements for m2.

Matrix Operations:Check if m1 and m2 are compatible for operations (using == operator).If compatible:Create m3 by adding m1 and m2 (using + operator).Create m4 by subtracting m2 from m1 (using - operator).Display the result of m1 + m2.Display the result of m1 - m2.If not compatible, display an error message.

Cleanup:The MATRIX class destructor should deallocate the dynamically allocated memory.

SOURCE CODE:-

```
#include <iostream>
using namespace std;
// Creating the class MATRIX
class MATRIX {
protected:
   int row;
   int col;
   int **matrix;
public:
   MATRIX(int a, int b) {  // Constructor to take row and column input
      row = a;
      col = b;
      matrix = new int*[row];
      for (int i = 0; i < row; ++i) {
         matrix[i] = new int[col];
      }
   }
   // Member function for filling matrix elements
   void matrix_element() {
        cout << "Enter matrix elements" << endl;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                cin >> matrix[i][j];
            }
        }
    }
    // Operator overloading for ==
    bool operator==(const MATRIX& other) const {
        return (row == other.row) && (col == other.col);
    }
    // Operator overloading for +
    MATRIX operator+(const MATRIX& other) const {
        MATRIX result(row, col);
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                result.matrix[i][j] = matrix[i][j] + other.matrix[i][j];
            }
        }
        return result;
    }
    // Operator overloading for -
    MATRIX operator-(const MATRIX& other) const {
        MATRIX result(row, col);
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                result.matrix[i][j] = matrix[i][j] - other.matrix[i][j];
            }
        }
        return result;
    }
    // Overload << operator to display the matrix
    friend ostream& operator<<(ostream& os, const MATRIX& mat) {
        for (int i = 0; i < mat.row; i++) {
```

```
        for (int j = 0; j < mat.col; j++) {
            os << mat.matrix[i][j] << " ";
        }
        os << "\n";
    }
    return os;
  }
  // Destructor to deallocate the memory given to matrix
  ~MATRIX() {
    for (int i = 0; i < row; ++i) {
        delete[] matrix[i];
    }
    delete[] matrix;
  }
};
// Main function
int main() {
  int i, j;
  cout << "Enter the number of rows and columns of 1st matrix" << endl;
  cin >> i >> j;
  // Creating object m1
  MATRIX m1(i, j);
  m1.matrix_element();
  cout << "Enter the number of rows and columns of 2nd matrix" << endl;
  cin >> i >> j;
  // Creating object m2
  MATRIX m2(i, j);
  m2.matrix_element();
  if (m1 == m2) { // Calling overloaded operator ==
    // Creating and initializing objects m3 and m4
    MATRIX m3 = m1 + m2; // Calling overloaded operator +
    MATRIX m4 = m1 - m2; // Calling overloaded operator -

    cout << "Matrix m1 + m2:" << std::endl;
```

```
        cout << m3; // Calling overloaded operator <<

        cout << "Matrix m1 - m2:" << std::endl;
        cout << m4; // Calling overloaded operator <<
    } else {
        cout << "Matrices are not compatible for addition and subtraction." << std::endl;
    }

    return 0;
}
```

OUTPUT:-

Correct Output:

```
Enter the number of rows and columns of 1st matrix
2 2
Enter matrix elements
1 2
3 4
Enter the number of rows and columns of 2nd matrix
2 2
Enter matrix elements
5 6
7 8
Matrix m1 + m2:
6 8
10 12
Matrix m1 - m2:
-4 -4
-4 -4
```

Wrong Output:

```
Enter the number of rows and columns of 1st matrix
2 3
```

```
Enter matrix elements
1 2 3
4 5 6
Enter the number of rows and columns of 2nd matrix
2 2
Enter matrix elements
7 8
9 10
Matrices are not compatible for addition and subtraction.
```

Error Output:

```
Enter the number of rows and columns of 1st matrix
2 2
Enter matrix elements
a b
c d
Enter the number of rows and columns of 2nd matrix
2 2
Enter matrix elements
1 2
3 4
Matrix m1 + m2:
1 2
3 4
Matrix m1 - m2:
-1 -2
-3 -4
```

8. Define a class SET with Data members: array of int, int variable to indicate number of elements in a SET object; and Member functions: to read element of a SET object, to print elements of a SET object, to find union of 2 objects of SET using operator overloading (S3=S1+S2), to find intersection of 2 objects of SET using operator overloading (S4= S1*S2). S1, S2, S3 and S4 are objects of SET. Use this class in a main function to show the above operations.

AIM :-

To create a class SET with Data members: array of int, int variable to indicate number of elements in a SET object; and Member functions: to read element of a SET object, to print elements of a SET object, to find union of 2 objects of SET using operator overloading (S3=S1+S2), to find intersection of 2 objects of SET using operator overloading (S4= S1*S2). S1, S2, S3 and S4 are objects of SET. Use this class in a main function to show the above operations.

DEFINATION :-

It defines a class named SET that models sets of integers and demonstrates the implementation of set union and intersection using operator overloading. The program allows users to input elements for two sets and then performs the union and intersection of these sets.

SYNTAX:-

```
int main() {

   // Create SET objects

   SET $1(10), $2(10), $3(20), $4(10));

   $1.readElements();    // Read elements for $1 and $2

   $2.readElements();

   // Perform union and intersection

   $3 = $1 + $2;  // Union

   $4 = $1 * $2;  // Intersection

   // Display results

   $3.printElements();

   $4.printElements();

   return 0;

}
```

PROCEDURE: -

Define the SET class with private data members for the array of integers and the number of elements.
Implement a constructor to initialize the set and allocate memory.
Implement readElements() to read elements from user input.
Implement printElements() to display the elements of the set.
Overload the '+' operator to perform set union and the '*' operator to perform set intersection.
Implement a destructor to free memory.
In the main() function, create two sets ($1 and $2), read elements, and perform set union and intersection.
Print the union and intersection results.

SOURCE CODE:

```
#include <iostream>

using namespace std;

// Class definition for SET

class SET {

private:

   int* array;         // Pointer to an array of integers

   int numElements;    // Number of elements in the set

public:

   // Constructor: Initializes the set with a given size

   SET(int size) {

      array = new int[size];

      numElements = 0;

   }

   // Method to read elements from user input

   void readElements() {

      cout << "Enter the number of elements: ";

      cin >> numElements;

      cout << "Enter " << numElements << " elements: ";

      for (int i = 0; i < numElements; ++i) {
            cin >> array[i];
        }
    }
    // Method to print elements of the set
    void printElements() {
        cout << "Elements in the set: ";
        for (int i = 0; i < numElements; ++i) {
            cout << array[i] << " ";
        }
        cout << std::endl;
    }
    // Overloaded '+' operator for set union
    SET operator+(const SET& other) {
        SET result(numElements + other.numElements);
        for (int i = 0; i < numElements; ++i) {
            result.array[i] = array[i];
        }
        for (int i = 0; i < other.numElements; ++i) {
            bool found = false;
            for (int j = 0; j < numElements; ++j) {
                if (other.array[i] == array[j]) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                result.array[result.numElements++] = other.array[i];
            }
        }
        return result;
    }
    // Overloaded '*' operator for set intersection
    SET operator*(const SET& other) {
        SET result(numElements);
        for (int i = 0; i < numElements; ++i) {
            for (int j = 0; j < other.numElements; ++j) {
                if (array[i] == other.array[j]) {
                    result.array[result.numElements++] = array[i];
                    break;
                }
            }
        }
        return result;
    }
    // Destructor: Frees memory used by the set
    ~SET() {
        delete[] array;
    }
};
// Main function
int main() {
    SET $1(10), $2(10), $3(20), $4(10);
    cout << "Enter elements for $1:" << endl;
    $1.readElements();
    cout << "Enter elements for $2:" << endl;

    $2.readElements();

    $3 = $1 + $2;  // Perform union

    $4 = $1 * $2;  // Perform intersection

    cout << "Union of $1 and $2: ";

    $3.printElements();

    cout << "Intersection of $1 and $2: ";

    $4.printElements();

    return 0;

}
```

OUTPUT:-

Correct Output:

```
Enter elements for $1:
Enter the number of elements: 3
Enter 3 elements: 1 2 3
Enter elements for $2:
Enter the number of elements: 4
Enter 4 elements: 2 3 4 5
Union of $1 and $2: Elements in the set: 1 2 3 4 5
Intersection of $1 and $2: Elements in the set: 2 3
```

Wrong Output:

```
Enter elements for $1:
Enter the number of elements: 3
Enter 3 elements: 1 2 3
Enter elements for $2:
Enter the number of elements: 4
Enter 4 elements: 5 6 7 8
Union of $1 and $2: Elements in the set: 1 2 3 5 6 7 8
Intersection of $1 and $2: Elements in the set:
```

Error Output:

```
Enter elements for $1:
Enter the number of elements: 3
Enter 3 elements: 1 2 3
Enter elements for $2:
Enter the number of elements: 4
Enter 4 elements: a b c d
Error: Invalid input for elements.
```

9. Create an abstract base class EMPLOYEE with data members: Name, EmpID and BasicSal and a pure virtual function Cal_Sal().Create two derived classes MANAGER (with data members: DA and HRA and SALESMAN (with data members: DA, HRA and TA). Write appropriate constructors and member functions to initialize the data, read and write the data and to calculate the net salary. The main() function should create array of base class pointers/references to invoke overridden functions and hence to implement run-time polymorphism.

AIM: -

To Create an abstract base class EMPLOYEE with data members: Name, EmpID and BasicSal and a pure virtual function Cal_Sal().Create two derived classes MANAGER (with data members: DA and HRA and SALESMAN (with data members: DA, HRA and TA). Write appropriate constructors and member functions to initialize the data, read and write the data and to calculate the net salary. The main() function should create array of base class pointers/references to invoke overridden functions and hence to implement run-time polymorphism.

DEFINITION: -

It demonstrates the use of inheritance and polymorphism to show how code can be reused in situations where there are different types of the same category each with its own unique characteristics but sharing a common link with a base. The program defines a base class EMPLOYEE and two derived classes MANAGER and SALESMAN, each representing a different type of employee with specific attributes and salary calculations.

SYNTAX: -

```
class AbstractBaseClass {
public:
   // Pure virtual function
   virtual void pureVirtualFunction() = 0;
};
// Derived class that inherits from the abstract base class
class DerivedClass : public AbstractBaseClass {
```

```
public:

   // Override the pure virtual function

   void pureVirtualFunction() override {

      // ... implementation ...

   }

};
```

PROCEDURE: -

Define the base class EMPLOYEE with protected data members and pure virtual function Cal_Sal() for calculating salary.
Define derived classes MANAGER and SALESMAN inheriting from the base class, with additional attributes and their implementations of Cal_Sal().
In the main() function, create an array of base class pointers to store instances of derived classes.
Use these pointers to create instances of MANAGER and SALESMAN classes.
Loop through the array, read employee data, calculate and display their salary using polymorphism.
Delete the dynamically allocated instances to free memory.

SOURCE Code:

```
#include <iostream>

#include <string>

// Base class EMPLOYEE

class EMPLOYEE {

protected:

   std::string Name;

   int EmpID;

   float BasicSal;

public:

   // Constructor with initialization list

   EMPLOYEE(const std::string& name, int empID, float basicSal)

      : Name(name), EmpID(empID), BasicSal(basicSal) {}

   // Pure virtual function to calculate salary
```

```
    virtual float Cal_Sal() = 0;

    // Method to read employee data

    void ReadData() {

        std::cout << "Enter Name: ";

        std::cin >> Name;

        std::cout << "Enter EmpID: ";

        std::cin >> EmpID;

        std::cout << "Enter Basic Salary: ";

        std::cin >> BasicSal;

    }

    // Method to display employee data

    void DisplayData() {

        std::cout << "Name: " << Name << "\n";

        std::cout << "EmpID: " << EmpID << "\n";

        std::cout << "Basic Salary: " << BasicSal << "\n";

    }

};

// Derived class MANAGER

class MANAGER : public EMPLOYEE {

private:

    float DA;

    float HRA;

public:

    // Constructor with initialization list

    MANAGER(const std::string& name, int empID, float basicSal, float da, float hra)

        : EMPLOYEE(name, empID, basicSal), DA(da), HRA(hra) {}

    // Implementation of the Cal_Sal method for managers
    float Cal_Sal() override {
        return BasicSal + DA + HRA;
    }
};
// Derived class SALESMAN
class SALESMAN : public EMPLOYEE {
private:
    float DA;
    float HRA;
    float TA;
public:
    // Constructor with initialization list
    SALESMAN(const std::string& name, int empID, float basicSal, float da, float hra, float ta)
        : EMPLOYEE(name, empID, basicSal), DA(da), HRA(hra), TA(ta) {}
    // Implementation of the Cal_Sal method for salesmen
    float Cal_Sal() override {
        return BasicSal + DA + HRA + TA;
    }
};
int main() {
    const int numEmployees = 2;
    EMPLOYEE* employees[numEmployees];
    // Creating instances of derived classes using base class pointers
    employees[0] = new MANAGER("John Doe", 101, 5000.0, 1000.0, 800.0);
    employees[1] = new SALESMAN("Jane Smith", 102, 4000.0, 800.0, 600.0, 300.0);
    // Loop through employees to read data, calculate and display salary
    for (int i = 0; i < numEmployees; ++i) {
        employees[i]->ReadData();

        float netSalary = employees[i]->Cal_Sal();

        employees[i]->DisplayData();

        std::cout << "Net Salary: " << netSalary << "\n\n";

    }

    // Freeing memory and deleting instances

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

        delete employees[i];

    }

    return 0;

}
```

__OUTPUT :-__

Correct Output:

```
Enter Name: John Doe
Enter EmpID: 101
Enter Basic Salary: 5000
Name: John Doe
EmpID: 101
Basic Salary: 5000
Net Salary: 6800
Enter Name: Jane Smith
Enter EmpID: 102
Enter Basic Salary: 4000
Name: Jane Smith
EmpID: 102
Basic Salary: 4000
Net Salary: 5700
```

Wrong Output:

```
Enter Name: John Doe
Enter EmpID: 101
Enter Basic Salary: abc
Name: John Doe
EmpID: 101
Basic Salary: 0
Net Salary: 1800
Enter Name: Jane Smith
Enter EmpID: 102
Enter Basic Salary: 4000
Name: Jane Smith
EmpID: 102
Basic Salary: 4000
Net Salary: 5700
```

Error Output:

```
Enter Name: John Doe
Enter EmpID: 101
Enter Basic Salary: 5000
Name: John Doe
EmpID: 101
Basic Salary: 5000
Net Salary: 6800
Enter Name: Jane Smith
Enter EmpID: 102
Enter Basic Salary: 4000
Name: Jane Smith
EmpID: 102
Basic Salary: 4000
Net Salary: 5700
```

10. Write a program to concatenate 2 strings using STL String class functions.

AIM :-

To create a program to concatenate 2 strings using STL String class functions.

DEFINITION :-

String concatenation in C++ STL involves merging two or more strings to form a single string. This can be done using the + operator or the append() function. The + operator combines strings, while append() appends one string to another. These functions simplify the process of joining strings, whether it's for creating sentences, displaying messages, or building complex textual data. String concatenation is essential for text manipulation and formatting, providing a versatile and efficient way to construct meaningful and coherent output in various programming scenarios.

SYNTAX:-

```
string append(const string& str);
```

PROCEDURE:-

**Declare two string variables.
Input the first string using getline() to allow spaces.
Input the second string using getline().
Create a temporary string and use the append() function to concatenate the two strings.
Print the concatenated string.**

SOURCE CODE:

```
#include <iostream>

#include <string>

using namespace std;

int main() {

   string str1, str2;

   cout << "Enter the first string: ";

   getline(cin, str1); // Allowing spaces in input

   cout << "Enter the second string: ";

   getline(cin, str2);

   string temp=str1;
```

```
    temp.append(str2); // Using the append() function to concatenate strings

   cout << "Concatenated string: " << temp << endl;

   return 0;

}
```

OUTPUT:-

Normal Output:

```
Enter the first string: Hello,
Enter the second string: world!
Concatenated string: Hello,world!
```

Wrong Output:

```
Enter the first string: Hello,
Enter the second string: world!
Concatenated string: Hello,sppend(world!)
```

Error Output:

```
Enter the first string: Hello
Enter the second string: World
Concatenated string: HelloWorld
Segmentation fault (core dumped)
```