Objective:

To understand OOP concepts in C++ such as using objects as function arguments, returning objects, working with arrays and pointers to objects, dynamic memory management, static and constant members, and friend functions/classes.

Tools and Libraries Used:

1) Programming Language: C++
2) IDE: Visual Studio Code
3) Libraries: #include<iostream>, #include<string>

Theory:

1. Object as Function Argument

In C++, objects of a class can be passed to functions in three ways:

a. Pass by value:

The function receives a copy of the object (copy constructor called). Modifications inside the function don't change the original object. It is Less efficient for large objects.

Syatax:

```
void functionName(ClassName obj);
```

example code:

```
#include <iostream>
using namespace std;
class Book {
public:
    string title;
    int pages;

    Book(string t, int p) : title(t), pages(p) {}

    void reducePages(Book b) {  // pass by value, copies object
        b.pages -= 10;
        cout << "Inside function: " << b.title << " has " << b.pages << " pages\n";
    }

    void show() {
        cout << "Book: " << title << ", Pages: " << pages << endl;
    }
};
```

```
int main() {
    Book myBook("C++ Guide", 300);
    cout << "Before function call:\n";
    myBook.show();
    myBook.reducePages(myBook);
    cout << "After function call:\n";
    myBook.show();
    return 0;
}
```

b. By reference:

When an object is passed to a function by reference, the function receives a direct reference (alias) to the original object using the & symbol. Any modifications made to the object inside the function will directly affect the original object outside the function.

Syntax:

```
void functionName(ClassName &obj);
```

example code:

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

class Employee {
public:
    string name;
    double salary;

    Employee(string n, double s) : name(n), salary(s) {}

    void giveRaise(Employee &emp) {
        emp.salary += 500.0;  // Increase salary by 500
        cout << "Inside function: " << emp.name << "'s salary = $" << emp.salary << endl;
    }
};

int main() {
    Employee e1("Alice", 4500.0);

    e1.giveRaise(e1);  // Pass object by reference

    cout << "After function call: " << e1.name << "'s salary = $" << e1.salary << endl;

    return 0;
}
```

c. By pointer:

Instead of passing the object directly, a pointer to the object is passed to the function. This allows indirect access to the object's members via the pointer (-> operator). Passing by pointer is especially useful when working with dynamic memory or when you want to allow the possibility of passing a nullptr (null pointer) for safety checks.

Syntax:

```
void functionName(ClassName *obj);
```

example code:

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

class Car {
public:
    string model;
    int mileage;

    Car(string m, int mil) : model(m), mileage(mil) {}

    void updateMileage(Car *c) {
        if (c != nullptr) {
            c->mileage += 100;  // Increase mileage by 100
            cout << "Inside function: " << c->model << "'s mileage = " << c->mileage << " km"
<< endl;
        }
    }

    void display() const {
        cout << "Car Model: " << model << ", Mileage: " << mileage << " km" << endl;
    }
};

int main() {
    Car car1("Tesla Model 3", 12000);

    cout << "Before function call:" << endl;
    car1.display();

    car1.updateMileage(&car1);  // Pass address of car1

    cout << "After function call:" << endl;
    car1.display();

    return 0;
}
```

2. Returning Objects from Functions in C++

Functions in C++ can return entire objects, just like they return primitive data types such as int or float. This feature is useful when you want a function to perform operations and return a new object representing the result, rather than modifying the original object.

Syntax:

```
class ClassName {
    // Data members and methods

public:
    ClassName functionName() {
        ClassName obj;
        // Initialize or process obj
        return obj;  // Return a new object
    }
};
```

Example code:

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

class Rectangle {
public:
    int width, height;

    Rectangle(int w = 0, int h = 0) : width(w), height(h) {}

    // Function that returns a new Rectangle object with doubled dimensions
    Rectangle doubleSize() {
        return Rectangle(width * 2, height * 2);
    }

    void display() const {
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

int main() {
    Rectangle rect1(5, 3);
    cout << "Original rectangle:" << endl;
    rect1.display();

    Rectangle rect2 = rect1.doubleSize();  // Returns a new object with doubled size
    cout << "New rectangle after doubling size:" << endl;
    rect2.display();

    return 0;
}
```

3. Array of Objects in C++

An array of objects is a collection of multiple instances of the same class stored consecutively in memory. It helps manage and organize multiple objects under a single array variable. This is especially useful when handling data for many similar entities like students, employees, or books, allowing you to easily process them using loops.

Syntax:

```
ClassName objectArray[arraySize];
```

Example code:

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

class Student {
    int roll;
    string name;

public:
    // Function to input student details
    void input() {
        cout << "Enter roll number: ";
        cin >> roll;
        cout << "Enter name: ";
        cin >> name;
    }

    // Function to display student details
    void display() const {
        cout << "Roll: " << roll << ", Name: " << name << endl;
    }
};

int main() {
    Student students[5];  // Array of 5 Student objects

    // Input details for each student
    for (int i = 0; i < 5; i++) {
        cout << "\nEnter details for student " << (i + 1) << ":\n";
        students[i].input();
    }

    // Display details of all students
    cout << "\nStudent Details:\n";
    for (int i = 0; i < 5; i++) {
        students[i].display();
    }

    return 0;
}
```

4. Pointer to Objects in C++

In C++, pointers can also point to class objects—just like they point to basic data types. These are known as pointers to objects. Once a pointer holds the address of an object, you can access that object's members using the -> (arrow) operator. Object pointers are helpful in dynamic memory management and when working with polymorphism and dynamic binding.

Syntax:

```
ClassName *ptr;         // Declare a pointer to ClassName
ptr = &object;          // Assign address of an object
ptr->memberFunction(); // Access members using pointer
```

example code:

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

class Student {
private:
    string name;
    int age;

public:
    void setData(string n, int a) {
        name = n;
        age = a;
    }

    void displayData() const {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main() {
    Student s1;
    Student *ptr;     // Declare a pointer to Student

    ptr = &s1;        // Store the address of s1 in the pointer

    ptr->setData("Rita", 20);   // Set data using pointer
    ptr->displayData();         // Display data using pointer

    return 0;
}
```

5. Dynamic Memory Allocation for Objects in C++

Dynamic memory allocation means assigning memory to an object at runtime, rather than at compile time. In C++, you use the new operator to allocate memory for an object dynamically, and delete to free that memory when it's no longer needed.

Syntax:

```
ClassName* ptr = new ClassName();   // Allocate memory dynamically
ptr->memberFunction();              // Access object members
delete ptr;                         // Free memory
```

example code:

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

class Student {
private:
    string name;
    int age;

public:
    void setData(string n, int a) {
        name = n;
        age = a;
    }

    void displayData() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Create object dynamically
    Student* s = new Student();

    string n;
    int a;

    cout << "Enter student name: ";
    getline(cin, n);
    cout << "Enter age: ";
    cin >> a;

    s->setData(n, a);    // Set data using pointer
    s->displayData();    // Display data using pointer

    delete s;            // Free dynamically allocated memory

    return 0;
}
```

6. Dynamic Memory Allocation for Object Arrays in C++

Dynamic memory allocation for object arrays allows you to create multiple objects at runtime based on user input or program conditions. Instead of using a fixed-size (static) array, memory is allocated using the new operator, which enables flexible and efficient memory use. Each object in the array can be accessed and manipulated just like elements in a regular array.

Syntax:

```
ClassName* ptr = new ClassName[size];   // Dynamically allocate array
ptr[i].memberFunction();                // Access elements using array indexing
delete[] ptr;                           // Free the allocated memory
```

example code:

```
#include <iostream>
using namespace std;
class Student {
private:
    string name;
    int age;
public:
    void setData(string n, int a) {
        name = n;
        age = a;
    }
    void displayData() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    int n;
    cout << "Enter number of students: ";
    cin >> n;

    // Dynamically allocate memory for n students
    Student* students = new Student[n];
    string name;
    int age;
    // Input data for each student
    for (int i = 0; i < n; i++) {
        cin.ignore();  // Clear input buffer before getline
        cout << "\nEnter name of student " << (i + 1) << ": ";
        getline(cin, name);
        cout << "Enter age of student " << (i + 1) << ": ";
        cin >> age;

        students[i].setData(name, age);  // Store data in object
    }
    // Display all student details
    cout << "\nStudent Details:\n";
    for (int i = 0; i < n; i++) {
        students[i].displayData();
    }
    // Free allocated memory
    delete[] students;
    return 0;
}
```

7. Dynamic Constructors in C++

A dynamic constructor is a constructor that uses the new keyword to allocate memory at runtime. This allows creating objects or variables whose size or values are not known at compile time.

Syntax:

```
class ClassName {
    datatype* pointer;
public:
    ClassName(parameters) {
        pointer = new datatype[size]; // dynamic memory allocation
    }
    ~ClassName() {
        delete[] pointer; // release memory
    }
};
```

Example code:

```
#include <iostream>
#include <string>
using namespace std;
class Book {
private:
    string* title;
    double* price;
public:
    // Dynamic Constructor
    Book(string t, double p) {
        title = new string(t);      // Allocate memory dynamically
        price = new double(p);
        cout << "Book object created dynamically!\n";}
    // Destructor to free dynamic memory
    ~Book() {
        delete title;
        delete price;
        cout << "Memory freed for Book object!\n";}
    // Display function
    void display() const {
        cout << "\nBook Details:\n";
        cout << "Title: " << *title << endl;
        cout << "Price: Rs. " << *price << endl;}
};
int main() {
    string userTitle;
    double userPrice;
    cout << "Enter book title: ";
    getline(cin, userTitle);
    cout << "Enter book price (Rs.): ";
    cin >> userPrice;
    // Create Book object dynamically
    Book* myBook = new Book(userTitle, userPrice);
    myBook->display();
    delete myBook; // Free allocated memory
    return 0;
}
```

8. this Pointer in C++

The this pointer is an implicit pointer available in all non-static member functions. It stores the address of the object that invoked the member function. Automatically passed by the compiler—no need to declare it manually. It's especially useful when the function parameters have the same name as the class data members.

Syntax:

```
this->memberName;
```

example code:

```
#include <iostream>
#include <string>
using namespace std;
class Book {
private:
    string title;
    double price;

public:
    // Member function with parameters having the same name as data members
    void setData(string title, double price) {
        // 'this' pointer is used to distinguish between class members and parameters
        this->title = title;
        this->price = price;
    }

    void display() const {
        // Accessing data members using 'this' is optional here
        cout << "Title: " << this->title << ", Price: $" << this->price << endl;
    }
};

int main() {
    Book b;
    string t;
    double p;

    // Take user input
    cout << "Enter book title: ";
    getline(cin, t);
    cout << "Enter book price: ";
    cin >> p;

    // Set data using member function
    b.setData(t, p);

    // Display book information
    b.display();

    return 0;
}
```

9. Static Data Members in C++

A static data member belongs to the class itself, not to any specific object. All objects share a single copy of the static variable. Declared using the static keyword inside the class and defined separately outside the class.

Syntax:

```
class ClassName {
    static data_type variable_name; // Declaration
};

// Definition outside the class
data_type ClassName::variable_name = value;
```

example code:

```
#include <iostream>
using namespace std;
class Employee {
private:
    static int employeeCount; // Static data member
    int id;
    string name;
public:
    Employee(int empId, string empName) {
        id = empId;
        name = empName;
        employeeCount++; // Increment static count on object creation
    }

    void display() const {
        cout << "ID: " << id << ", Name: " << name << endl;
    }

    // Static function to access static member
    static void showTotalEmployees() {
        cout << "Total Employees: " << employeeCount << endl;
    }
};

// Definition of static member outside the class
int Employee::employeeCount = 0;

int main() {
    Employee e1(101, "Alice");
    Employee e2(102, "Bob");

    e1.display();
    e2.display();
    // Access static member via static function
    Employee::showTotalEmployees();
    return 0;
}
```

10. Static Member Functions in C++

A static member function is a function that belongs to the class itself, not to any object. Declared using the static keyword inside the class. It can be called without creating an object of the class.

Syntax:

```
class ClassName {
public:
    static return_type functionName(parameters); // Declaration
};

// Definition outside the class
return_type ClassName::functionName(parameters) {
    // Function body
}
```

example code:

```
#include <iostream>
using namespace std;
class Product {
private:
    static int productCount;  // Static data member to count products
    string name;
    double price;
public:
    Product(string n, double p) : name(n), price(p) {
        productCount++;  // Increment count whenever a new object is created
    }

    void display() const {
        cout << "Product: " << name << ", Price: $" << price << endl;
    }

    // Static member function
    static void showProductCount() {
        cout << "Total Products Created: " << productCount << endl;
    }
};
// Definition of the static data member
int Product::productCount = 0;

int main() {
    Product p1("Laptop", 80000);
    Product p2("Smartphone", 35000);

    p1.display();
    p2.display();

    // Call static member function without using any object
    Product::showProductCount();
    return 0;
}
```

11. Constant Member Functions in C++

A constant member function ensures that no changes are made to the object's data members when the function is called.

Syntax:

```
class ClassName {
public:
    returnType functionName() const; // Declaration
};

returnType ClassName::functionName() const {
    // Read-only access, no modifications to members
}
```

Example code:

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

class Book {
private:
    string title;
    double price;

public:
    // Constructor to initialize data members
    Book(string t, double p) : title(t), price(p) {}

    // Constant member function (read-only access)
    void display() const {
        cout << "Title: " << title << ", Price: ₹" << price << endl;
    }
};

int main() {
    const Book b1("C++ Fundamentals", 499.99); // Constant object

    b1.display(); // Allowed: display() is a const member function

    return 0;
}
```

12. Constant Objects in C++

A constant object in C++ is an instance of a class that cannot be modified after it is created. It is declared using the const keyword and can only invoke constant member functions.

Syntax:

```
const ClassName objectName(arguments);
```

example code:

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

class Student {
private:
    int roll;
    string name;

public:
    // Constructor to initialize object
    Student(int r, string n) : roll(r), name(n) {}

    // Const member function for read-only access
    void display() const {
        cout << "Roll No: " << roll << ", Name: " << name << endl;
    }

    // Non-const function: modifies object (not allowed on const object)
    void updateRoll(int r) {
        roll = r;
    }
};

int main() {
    const Student s1(101, "Kiran"); // Constant object

    s1.display();

    return 0;
}
```

13. Friend Function in C++

In object-oriented programming, data hiding ensures that internal class details (private/protected members) are inaccessible directly from outside the class.
However, sometimes a non-member function needs access to these private members. In such cases, we use a friend function.

Syntax:

```
class ClassName {
private:
    data_type member;
public:
    friend return_type functionName(ClassName obj); // Friend declaration
};

return_type functionName(ClassName obj) {
    // Can access obj.member (even if private)
}
```

Example code:

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

class Student {
private:
    int rollNo;
    int marks;

public:
    // Constructor to initialize student data
    Student(int r, int m) {
        rollNo = r;
        marks = m;
    }

    // Friend function declaration
    friend void displayInfo(Student s);
};

// Friend function definition
void displayInfo(Student s) {
    // Accessing private members directly
    cout << "Roll Number: " << s.rollNo << endl;
    cout << "Marks: " << s.marks << endl;
}

int main() {
    Student s1(101, 90);      // Creating a Student object
    displayInfo(s1);          // Calling friend function

    return 0;
}
```

14. Friend Class in C++

A friend class is a class that is allowed to access private and protected members of another class. When Class A declares Class B as a friend, all member functions of Class B can access the private and protected members of Class A. Friendship is not mutual by default. If Class A declares Class B as friend, Class B can access A's private members, but Class A cannot access Class B's private members unless B also declares A as a friend.

Syntax:

```
class ClassB;  // Forward declaration

class ClassA {
    friend class ClassB;  // ClassB is friend of ClassA
private:
    int dataA;
public:
    ClassA(int x) : dataA(x) {}
};
class ClassB {
public:
    void showData(ClassA &a) {
        // Can access private member of ClassA because ClassB is a friend
        cout << "Data from ClassA: " << a.dataA << endl;
    }
};
```

Example code:

```
#include <iostream>
using namespace std;
class Account; // Forward declaration
class Bank {
public:
    void showBalance(Account &acc);
};

class Account {
private:
    double balance;
public:
    Account(double b) {
        balance = b;
    }
    friend class Bank;  // Bank class is friend of Account
};
void Bank::showBalance(Account &acc) {
    // Bank class can access private members of Account
    cout << "Account balance is: $" << acc.balance << endl;
}
int main() {
    Account myAccount(2500.75);
    Bank myBank;
    myBank.showBalance(myAccount);  // Display private balance via friend class
    return 0;
}
```

Q1) Write a program to define a class that uses static data members and static member functions to count and display the number of objects created.

Code:

```
#include <iostream>
using namespace std;
class Counter {
private:
   static int count;

public:
   Counter() {
      count++;
   }

   ~Counter() {
      count--;
   }

   static void displayCount() {
      cout << "No of objects: " << count << endl;
   }
};

int Counter::count = 0;

int main() {
   Counter::displayCount();
   Counter c1;
   Counter::displayCount();
   Counter c2;
   Counter::displayCount();
   Counter c3;
   Counter::displayCount();
   return 0;
}
```

Output:

```
No of objects: 0
No of objects: 1
No of objects: 2
No of objects: 3
```

Q2) Write a program to create a class that uses a copy constructor to copy data from one object to another.

Code:

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

class Student {
private:
   string name;
   int age;
   float gpa;

public:
   Student() {
      name = "Someone";
      age = 0;
      gpa = 0.0;
   }
   Student(string n, int a, float g) {
      name = n;
      age = a;
      gpa = g;
   }
   Student(const Student &s) {
      name = s.name;
      age = s.age;
      gpa = s.gpa;
   }
   void display() {
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
      cout << "GPA: " << gpa << endl;
   }
};
int main() {
   Student student1("Student 1", 20, (float)3.8);
   cout << "Student 1:" << endl;
   student1.display();
   cout << endl;
   Student student2 = student1;
   cout << "Student 2 details (copied):" << endl;
   student2.display();
   return 0;
}
```

Output:

```
Student 1:
Name: Student 1
Age: 20
GPA: 3.8

Student 2 details (copied):
Name: Student 1
Age: 20
GPA: 3.8
```

Q3) Write a program to dynamically allocate and deallocate memory for a single object and an array of objects using the new and delete operators.

Code:

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

class Student {

   private:
      string name;
      int age;

   public:
Student() : name(""), age(0) {}

      void setData() {
         cout << "Enter name: ";
         cin >> name;
         cout << "Enter age: ";
         cin >> age;
      }
      void display() {
         cout << "Name: " << name << endl;
         cout << "Age: " << age << endl;
      }
};

int main() {
   Student* singleStudent = new Student();

   singleStudent->setData();
   singleStudent->display();

   delete singleStudent;
   int size;
   cout << "Enter number of students: ";
   cin >> size;
   Student* studentArray = new Student[size];
   cout << "Enter data for " << size << " students:" << endl;
   for(int i = 0; i < size; i++) {
      cout << "Student " << i+1 << ":" << endl;
      studentArray[i].setData();
   }
   cout << "\n Displaying:" << endl;
   for(int i = 0; i < size; i++) {
      cout << "Student " << i+1 << ":" << endl;
      studentArray[i].display();
   }

   delete[] studentArray;

   return 0;
}
```

Output:

```
Enter name: ram
Enter age: 12
Name: ram
Age: 12
Enter number of students: 2
Enter data for 2 students:
Student 1:
Enter name: sita
Enter age: 15
Student 2:
Enter name: hari
Enter age: 14

 Displaying:
Student 1:
Name: sita
Age: 15
Student 2:
Name: hari
Age: 14
```

Q4) Write a program that demonstrates default, parameterized, and copy constructors. Use the this pointer to calculate the midpoint between two points.

Code:

```
#include <iostream>
using namespace std;
class Point {
private:
   double x, y;
public:
   Point() {
      this->x = 0.0;
      this->y = 0.0;
   }
   Point(double x1, double y1) {
      this->x = x1;
      this->y = y1;
   }
   Point(const Point& p) {
      this->x = p.x;
      this->y = p.y;
   }
   Point getMidpoint(const Point& p) {
      Point midpoint;
      midpoint.x = (this->x + p.x) / 2.0;
      midpoint.y = (this->y + p.y) / 2.0;
      return midpoint;
   }
   void display() {
      cout << "(" << this->x << ", " << this->y << ")" << endl;
   }
};
int main() {
   Point p1;
   cout << "Point 1 (default constructor): ";
   p1.display();
   Point p2(5.0, 3.0);
   cout << "Point 2 (parameterized constructor): ";
   p2.display();
   Point p3(p2);
   cout << "Point 3 (copy constructor): ";
   p3.display();
   Point p4(1.0, 2.0);
   cout << "Point 4 (parameterized constructor): ";
   p4.display();
   Point midpoint = p2.getMidpoint(p4);
   cout << "Midpoint between p2 and p4: ";
   midpoint.display();
   return 0;
}
```

Output:

```
Point 1 (default constructor): (0, 0)
Point 2 (parameterized constructor): (5, 3)
Point 3 (copy constructor): (5, 3)
Point 4 (parameterized constructor): (1, 2)
Midpoint between p2 and p4: (3, 2.5)
```

Q5) Write a program to define a function that takes an object as a reference parameter and modifies its data members.

Code:

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

class Rectangle {
private:
   double length;
   double width;

public:
   Rectangle(double l = 0, double w = 0) : length(l), width(w) {}

   void display() const {
      cout << "Length: " << length << ", Width: " << width << endl;
   }

   friend void modifyRectangle(Rectangle& rect, double l, double w);
};

void modifyRectangle(Rectangle& rect, double l, double w) {
   rect.length += l;
   rect.width += w;
}

int main() {
   Rectangle rect(5, 3);

   cout << "Original Rectangle:" << endl;
   rect.display();

   double l, w;
   cout << "Enter length to add: ";
   cin >> l;
   cout << "Enter width to add: ";
   cin >> w;

   modifyRectangle(rect, l, w);

   cout << "Modified Rectangle:" << endl;
   rect.display();

   return 0;
}
```

Output:

```
Original Rectangle:
Length: 5, Width: 3
Enter length to add: 4
Enter width to add: 7
Modified Rectangle:
Length: 9, Width: 10
```

Q6) Write a program that defines a class inside a namespace and uses a reference to modify object attributes.

Code:

```
#include <iostream>
using namespace std;
namespace StdMgmt {
   class Std {
   private:
      string name;
      int age;

   public:
      Std(string n, int a) : name(n), age(a) {}

      void updateInfo(string& newName, int& newAge) {
         name = newName;
         age = newAge;
      }

      void displayInfo() const {
        cout << "Name: " << name << std::endl;
        cout << "Age: " << age << std::endl;
      }
   };
}

int main() {
   using namespace StdMgmt;

   Std std("Someone", 20);

   cout << "Initial student information:" << endl;
   std.displayInfo();

   string newName;
   int newAge;
   cout << "Enter new name: ";
   cin >> newName;
   cout << "Enter new age: ";
   cin >> newAge;
   std.updateInfo(newName, newAge);

   cout << "Updated student information:" << endl;
   std.displayInfo();

   return 0;
}
```

Output:

```
Initial student information:
Name: Someone
Age: 20
Enter new name: glaive
Enter new age: 23
Updated student information:
Name: glaive
Age: 23
```

Q7) Write a program that defines a constant member function to access constant object data, and use const_cast to modify it safely.

Code:

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

class Info {
private:
   int value;

public:
   Info(int val) : value(val) {}

   void printValue() const {
      cout << "Current value: " << value << endl;
   }

   void modifyValue() const {
      Info* modifiableThis = const_cast<Info*>(this);
      modifiableThis->value += 10;
   }
};

int main() {
   const Info obj(15);

   obj.printValue();

   const_cast<Info&>(obj).modifyValue();

   obj.printValue();

   return 0;
}
```

Output:

```
Current value: 15
Current value: 25
```

Q8) Write a program to demonstrate a friend function that accesses private data from two different classes and adds their values.

Code:

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

class Alpha;
class Beta;

class Alpha {
private:
    int numA;

public:
    Alpha(int val) : numA(val) {}

    friend int addValues(const Alpha&, const Beta&);
};

class Beta {
private:
    int numB;

public:
    Beta(int val) : numB(val) {}

    friend int addValues(const Alpha&, const Beta&);
};

int addValues(const Alpha& a, const Beta& b) {
    return a.numA + b.numB;
}

int main() {
    Alpha obj1(10);
    Beta obj2(25);

    int total = addValues(obj1, obj2);

    cout << "Sum of values from both objects: " << total << endl;

    return 0;
}
```

Output:

```
Sum of values from both objects: 35
```

Q9) Write a program to create a class LandMeasure with units Ropani, Ana, Paisa, and Dam. Define a function to add two objects, applying proper unit conversion (16 Ana = 1 Ropani, etc.).

Code:

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

class LandMeasure {
private:
    int ropani, ana, paisa, dam;

public:
    LandMeasure(int r = 0, int a = 0, int p = 0, int d = 0)
        : ropani(r), ana(a), paisa(p), dam(d) {}

    LandMeasure add(const LandMeasure& other) {
        int totalDam = dam + other.dam;
        int totalPaisa = paisa + other.paisa + totalDam / 4;
        totalDam %= 4;

        int totalAna = ana + other.ana + totalPaisa / 4;
        totalPaisa %= 4;

        int totalRopani = ropani + other.ropani + totalAna / 16;
        totalAna %= 16;

        return LandMeasure(totalRopani, totalAna, totalPaisa, totalDam);
    }

    void display() const {
        cout << "Ropani: " << ropani << ", Ana: " << ana
             << ", Paisa: " << paisa << ", Dam: " << dam << endl;
    }
};

int main() {
    LandMeasure plot1(1, 15, 3, 3);
    LandMeasure plot2(2, 10, 2, 2);

    cout << "First Plot: ";
    plot1.display();

    cout << "Second Plot: ";
    plot2.display();

    LandMeasure totalPlot = plot1.add(plot2);

    cout << "Total Land: ";
    totalPlot.display();
    return 0;
}
```

Output:

```
First Plot: Ropani: 1, Ana: 15, Paisa: 3, Dam: 3
Second Plot: Ropani: 2, Ana: 10, Paisa: 2, Dam: 2
Total Land: Ropani: 4, Ana: 10, Paisa: 2, Dam: 1
```

Q10) Write a program to define a class String that uses a dynamic constructor to allocate memory and join two strings entered by the user.

Code:

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

class String {
private:
   char* str;

public:
   String() {
      str = new char[1];
      str[0] = '\0';
   }

   String(const char* s1, const char* s2) {
      int len = strlen(s1) + strlen(s2) + 1;
      str = new char[len];
      strcpy(str, s1);
      strcat(str, s2);
   }

   void display() const {
      cout << "Combined string: " << str << endl;
   }

   ~String() {
      delete[] str;
   }
};

int main() {
   char input1[100], input2[100];

   cout << "Enter first string: ";
   cin >> input1;

   cout << "Enter second string: ";
   cin >> input2;

   String combined(input1, input2);

   combined.display();

   return 0;
}
```

Output:

```
Enter first string: hello
Enter second string: world
Combined string: helloworld
```

Q11) Write a program to define a class Rectangle with length and width. Implement a member function that takes another Rectangle object and returns a new object with combined dimensions.

Code:

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

class Rectangle {
private:
   double length, width;

public:
   Rectangle(double l = 0, double w = 0) : length(l), width(w) {}

   Rectangle combine(const Rectangle& other) {
      return Rectangle(length + other.length, width + other.width);
   }

   void display() const {
      cout << "Length: " << length << ", Width: " << width << endl;
   }
};

int main() {
   Rectangle r1(4.5, 2.0);
   Rectangle r2(3.5, 1.5);

   cout << "Rectangle 1:" << endl;
   r1.display();

   cout << "Rectangle 2:" << endl;
   r2.display();

   Rectangle combined = r1.combine(r2);

   cout << "Combined Rectangle:" << endl;
   combined.display();

   return 0;
}
```

Output:

```
Rectangle 1:
Length: 4.5, Width: 2
Rectangle 2:
Length: 3.5, Width: 1.5
Combined Rectangle:
Length: 8, Width: 3.5
```

Q12) Write a program to create an array of five Employee objects, each with name and salary. Display the employee with the highest salary.

Code:

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

class Employee {
private:
   string name;
   double salary;

public:
   void setData() {
      cout << "Enter name: ";
      cin >> name;
      cout << "Enter salary: ";
      cin >> salary;
   }

   double getSalary() const {
      return salary;
   }

   void display() const {
      cout << "Name: " << name << ", Salary: " << salary << endl;
   }
};

int main() {
   Employee staff[2];

   for (int i = 0; i < 2; ++i) {
      cout << "Enter details for Employee " << i + 1 << ":" << endl;
      staff[i].setData();
   }

   int topIndex = (staff[0].getSalary() > staff[1].getSalary()) ? 0 : 1;

   cout << "\nEmployee with highest salary:" << endl;
   staff[topIndex].display();

   return 0;
}
```

Output:

```
Enter details for Employee 1:
Enter name: Sam
Enter salary: 45000
Enter details for Employee 2:
Enter name: Nina
Enter salary: 50000

Employee with highest salary:
Name: Nina, Salary: 50000
```

Q13) Write a program to define a class Point. Use pointers to dynamically allocate memory for two points and calculate the distance between them.

Code:

```
#include <iostream>
#include <cmath>
using namespace std;
class Point {
private:
   double x, y;
public:
   void setCoordinates() {
      cout << "Enter x: ";
      cin >> x;
      cout << "Enter y: ";
      cin >> y;
   }

   double getX() const {
      return x;
   }

   double getY() const {
      return y;
   }
};
double calculateDistance(Point* p1, Point* p2) {
   double dx = p1->getX() - p2->getX();
   double dy = p1->getY() - p2->getY();
   return sqrt(dx * dx + dy * dy);
}

int main() {
   Point* pointA = new Point;
   Point* pointB = new Point;

   cout << "Enter coordinates for Point A:" << endl;
   pointA->setCoordinates();

   cout << "Enter coordinates for Point B:" << endl;
   pointB->setCoordinates();

   double dist = calculateDistance(pointA, pointB);
   cout << "Distance between the two points: " << dist << endl;

   delete pointA;
   delete pointB;

   return 0;
}
```

Output:

```
Enter coordinates for Point A:
Enter x: 3
Enter y: 4
Enter coordinates for Point B:
Enter x: 0
Enter y: 0
Distance between the two points: 5
```

Q14) Write a program to define a class Box. Use the this pointer in a member function to compare two boxes and return the one with the greater volume.

Code:

```
#include <iostream>
using namespace std;
class Box {
private:
   double length, width, height;

public:
   Box(double l = 0, double w = 0, double h = 0) : length(l), width(w), height(h) {}

   double volume() const {
      return length * width * height;
   }

   Box compareVolume(const Box& other) {
      if (this->volume() > other.volume()) {
         return *this;
      } else {
         return other;
      }
   }

   void display() const {
      cout << "Dimensions: " << length << " x " << width << " x " << height << endl;
      cout << "Volume: " << volume() << endl;
   }
};
int main() {
   Box b1(3, 4, 5);
   Box b2(6, 2, 2);

   cout << "Box 1:" << endl;
   b1.display();

   cout << "Box 2:" << endl;
   b2.display();

   Box biggerBox = b1.compareVolume(b2);

   cout << "\nBox with greater volume:" << endl;
   biggerBox.display();
   return 0;
}
```

Output:

```
Box 1:
Dimensions: 3 x 4 x 5
Volume: 60
Box 2:
Dimensions: 6 x 2 x 2
Volume: 24

Box with greater volume:
Dimensions: 3 x 4 x 5
Volume: 60
```

Discussion:

We explored how to use objects as inputs and outputs in functions, which helps make our programs more modular. We also learned how to work with arrays and pointers to handle multiple objects efficiently. Managing memory dynamically showed us how to create and delete objects while the program runs, saving resources. Additionally, we studied static and constant members to control shared and unchangeable data, and friend functions to allow special access between classes.

Conclusion:

By understanding these core OOP concepts in C++, we can write more flexible, efficient, and organized programs. These tools let us better control how data and functions interact, making our code cleaner and easier to maintain.