# Experiment 1:Illustrate class and objects

Aim of the Program:

To demonstrate the concept of classes and objects in C++ by creating a Car class, defining its attributes (brand, model, year) and a method to display car information, and then creating multiple Car objects to store and display their unique data.

Algorithm:

Define the Car class:
Declare private data members: string brand, string model, int year.
Declare public member functions:
A constructor to initialize brand, model, and year.
A displayInfo() function to print the car's information.
Create objects of the Car class in main():
Instantiate multiple Car objects, passing different values to their constructors.
Access and display object data:
Call the displayInfo() method for each Car object to show its specific details.

Program:

```
#include <iostream>
#include <string>

class Car {
private:
    std::string brand;
    std::string model;
    int year;

public:
    // Constructor
    Car(std::string b, std::string m, int y) {
        brand = b;
        model = m;
        year = y;
    }

    // Member function to display car information
    void displayInfo() {
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year << std::endl;
    }
};

int main() {
    // Create Car objects
    Car car1("Toyota", "Camry", 2020);
    Car car2("Honda", "Civic", 2022);
    Car car3("Ford", "Mustang", 1969);

    // Display information for each car object
    std::cout << "Car 1 Information:" << std::endl;
    car1.displayInfo();

    std::cout << "Car 2 Information:" << std::endl;
    car2.displayInfo();

    std::cout << "Car 3 Information:" << std::endl;
    car3.displayInfo();
```

```
return 0;
}
```

Output:

```
Car 1 Information:
Brand: Toyota, Model: Camry, Year: 2020
Car 2 Information:
Brand: Honda, Model: Civic, Year: 2022
Car 3 Information:
Brand: Ford, Model: Mustang, Year: 1969
```

Result:

The program successfully demonstrates the use of classes and objects in C++. The Car class acts as a blueprint, and car1, car2, and car3 are distinct objects, each holding its own set of data (brand, model, year) and capable of performing the displayInfo() action defined in the class. This illustrates how objects encapsulate data and behavior, a core principle of object-oriented programming

# Experiment 2: Implement member function defined inside and outside the class

Aim:

The aim of this program is to demonstrate the implementation of member functions both inside and outside a C++ class definition. This illustrates different approaches to defining class behavior and the use of the scope resolution operator (::) when defining functions outside the class.

Algorithm:

Define a Class: Create a C++ class, for example, MyClass.

Declare Member Variables: Declare private member variables within the class to store data.
Define Member Function Inside Class:
Declare and define a member function directly within the class definition. This function will have direct access to the class's member variables.

Declare Member Function Outside Class:
Declare a member function prototype within the class definition.
Define this member function outside the class definition using the scope resolution operator (::) to specify which class it belongs to.

Create Objects: In the main function, create an object of the defined class.
Call Member Functions: Call both the inside-defined and outside-defined member functions using the created object.

Program:

```
#include <iostream>

class MyClass {
private:
    int data1;
    int data2;

public:
    // Member function defined inside the class
    void setDataInside(int val1) {
        data1 = val1;
        std::cout << "Data1 set inside class: " << data1 << std::endl;
    }

    // Member function declared inside, defined outside the class
    void setDataOutside(int val2);
    void displayData(); // Declared inside, defined outside
};

// Member function definition outside the class
void MyClass::setDataOutside(int val2) {
    data2 = val2;
    std::cout << "Data2 set outside class: " << data2 << std::endl;
}

// Another member function definition outside the class
void MyClass::displayData() {
    std::cout << "Displaying data: Data1 = " << data1 << ", Data2 = " << data2 << std::endl;
}

int main() {
    MyClass obj; // Create an object of MyClass

    // Call member function defined inside the class
    obj.setDataInside(10);

    // Call member function defined outside the class
    obj.setDataOutside(20);

    // Call another member function defined outside the class to display data
    obj.displayData();

    return 0;
}
```

Output:

```
Data1 set inside class: 10
Data2 set outside class: 20
Displaying data: Data1 = 10, Data2 = 20
```

Result:

The program successfully demonstrates the implementation of member functions both inside and outside a C++ class. The setDataInside function, defined directly within the MyClass definition, and the setDataOutside and displayData functions, declared inside but defined outside using the scope resolution operator, all function correctly. This illustrates how class behavior can be defined using these two distinct methods, achieving the same functional outcome.

# Experiment 3:To demonstrate the concept of function overloading applied to the member functions

Aim:

The aim is to demonstrate function overloading in C++ by defining multiple member functions with the same name but different parameter lists within a class, and then invoking the appropriate function based on the arguments provided during the function call.

Algorithm:

Define a class: Create a class (e.g., Calculator) that will contain the overloaded member functions.

Declare overloaded member functions: Inside the class, declare multiple functions with the same name (e.g., add), but with different parameter lists. The differences can be in the number of parameters, the types of parameters, or the order of parameters.

Implement the overloaded member functions: Provide the definitions for each overloaded function, where each function performs a similar operation but adapts to its specific parameter list.

Create an object: In the main function, create an object of the class.
Call overloaded functions: Invoke the overloaded member functions using the object, passing different sets of arguments to trigger the execution of the appropriate overloaded version.

Program:

```
#include <iostream>

class Calculator {
public:
    // Overloaded function to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded function to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded function to add two doubles
    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;

    // Call add with two integers
    int sum1 = calc.add(5, 10);
    std::cout << "Sum of two integers: " << sum1 << std::endl;

    // Call add with three integers
    int sum2 = calc.add(5, 10, 15);
    std::cout << "Sum of three integers: " << sum2 << std::endl;

    // Call add with two doubles
    double sum3 = calc.add(3.5, 7.2);
    std::cout << "Sum of two doubles: " << sum3 << std::endl;

    return 0;
}
```

Output:

Sum of two integers: 15
Sum of three integers: 30
Sum of two doubles: 10.7

Result

The program successfully demonstrates function overloading. The Calculator class contains three add member functions, each with a unique parameter list. When calc.add(5, 10) is called, the version taking two integers is executed. When calc.add(5, 10, 15) is called, the version taking three integers is executed. Finally, when calc.add(3.5, 7.2) is called, the version taking two doubles is executed. This illustrates that the compiler resolves the correct overloaded function based on the number and types of arguments provided during the function call, enabling a single function name to perform similar operations on different data types or with varying numbers of inputs.

# Experiment 4: Implement passing object as function arguments and return object from function

Aim:

The aim is to demonstrate passing an object as an argument to a function and returning an object from a function in a programming language, illustrating how object data can be manipulated and transferred between different parts of a program.

Algorithm:

Define a Class: Create a class with some data members and a constructor to initialize them.

Create Objects: In the main function (or equivalent), instantiate one or more objects of the defined class
.
Define a Function to Accept an Object: Create a function that takes an object of the defined class as an argument.

Define a Function to Return an Object: Create another function that performs some operation and returns a new object of the same class.

Pass Object to Function: Call the function defined in step 3, passing an object created in step 2 as an argument.

Receive and Use Returned Object: Call the function defined in step 4, and store the returned object in a new variable.

Display Results: Print the data members of the objects to verify the operations.

Program:

```
#include <iostream>

class MyClass {
public:
    int value1;
    double value2;

    // Constructor
    MyClass(int v1 = 0, double v2 = 0.0) : value1(v1), value2(v2) {}

    // Member function to display object data
    void display() const {
        std::cout << "Value 1: " << value1 << ", Value 2: " << value2 << std::endl;
    }
};

// Function that takes an object by value and returns a new object
MyClass addValues(MyClass obj1, MyClass obj2) {
    MyClass result(obj1.value1 + obj2.value1, obj1.value2 + obj2.value2);
    return result; // Returns a new MyClass object
```

```
}

// Function that creates and returns an object
MyClass createObject(int v1, double v2) {
    MyClass newObj(v1, v2);
    return newObj; // Returns a newly created MyClass object
}

int main() {
    // Create objects
    MyClass objA(10, 5.5);
    MyClass objB(20, 3.2);

    std::cout << "Original Objects:" << std::endl;
    std::cout << "Obj A: ";
    objA.display();
    std::cout << "Obj B: ";
    objB.display();

    // Pass objects as arguments and return a new object
    MyClass objC = addValues(objA, objB);
    std::cout << "\nObject C (result of addValues): ";
    objC.display();

    // Create and return an object from a function
    MyClass objD = createObject(50, 10.8);
    std::cout << "Object D (result of createObject): ";
    objD.display();

    return 0;
}
```

Output:

```
Original Objects:
Obj A: Value 1: 10, Value 2: 5.5
Obj B: Value 1: 20, Value 2: 3.2

Object C (result of addValues): Value 1: 30, Value 2: 8.7
Object D (result of createObject): Value 1: 50, Value 2: 10.8
```

Result:

The code successfully demonstrates passing objects as arguments to the addValues function and returning objects from both addValues and createObject functions in C++. The addValues function receives copies of objA and objB, performs an addition on their member values, and returns a new MyClass object (objC) containing the summed values. The createObject function constructs a MyClass object with specified values and returns it, which is then assigned to objD. The output clearly shows the initial states of objA and objB, and the final states of objC and objD, confirming the successful passing and returning of objects

# Experiment 5:To demonstrate the use of constructor with its types and destructor

Aim:

To illustrate the functionality of different types of constructors for object initialization and the role of destructors in object cleanup and resource deallocation.

Algorithm:

Define a Class: Create a class (e.g., MyClass) with private data members and public member functions, including constructors and a destructor.

Default Constructor: Implement a default constructor that takes no arguments and initializes data members to default values.

Parameterized Constructor: Implement a parameterized constructor that takes arguments to initialize data members with specific values.

Copy Constructor: Implement a copy constructor that takes an object of the same class as a reference and initializes a new object with the values of the existing object.

Destructor: Implement a destructor to perform cleanup operations when an object is destroyed.

Main Function:

Create an object using the default constructor.
Create an object using the parameterized constructor.
Create an object using the copy constructor.
Call member functions to display object data.
Observe the destructor calls when objects go out of scope.

Program:

```
#include <iostream>
#include <string>

class MyClass {
private:
    std::string name;
    int id;

public:
    // 1. Default Constructor
    MyClass() {
        name = "Default Name";
        id = 0;
        std::cout << "Default Constructor called for object: " << name << std::endl;
    }

    // 2. Parameterized Constructor
    MyClass(std::string n, int i) {
        name = n;
        id = i;
        std::cout << "Parameterized Constructor called for object: " << name << std::endl;
    }

    // 3. Copy Constructor
    MyClass(const MyClass& other) {
        name = "Copy of " + other.name;
        id = other.id;
        std::cout << "Copy Constructor called for object: " << name << std::endl;
    }

    // Member function to display data
```

```
std::cout << "Name: " << name << ", ID: " << id << std::endl;
    }

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called for object: " << name << std::endl;
    }
};

int main() {
    std::cout << "Creating objects..." << std::endl;

    // Using Default Constructor
    MyClass obj1;
    obj1.display();

    // Using Parameterized Constructor
    MyClass obj2("Parameterized Object", 101);
    obj2.display();

    // Using Copy Constructor
    MyClass obj3 = obj2; // or MyClass obj3(obj2);
    obj3.display();

    std::cout << "Objects created. Exiting main function..." << std::endl;
    return 0;
}
```

Output:

Creating objects…

Default Constructor called for object: Default Name

Name: Default Name, ID: 0

Parameterized Constructor called for object: Parameterized Object

Name: Parameterized Object, ID: 101

Copy Constructor called for object: Copy of Parameterized Object

Name: Copy of Parameterized Object, ID: 101

Objects created. Exiting main function…

Destructor called for object: Copy of Parameterized Object

Destructor called for object: Parameterized Object

Destructor called for object: Default Name

Result:

The output demonstrates the invocation of different constructor types based on how objects are created: obj1 triggers the default constructor, obj2 triggers the parameterized constructor, and obj3 triggers the copy constructor. Furthermore, the output clearly shows the destructor being called for each object in reverse order of their creation (when they go out of scope), confirming their role in performing cleanup operations. This example successfully illustrates the purpose and usage of constructors and destructors in object-oriented programming.

# Experiment-6: Ilustrate the use of static data member and static member function

Aim:

The aim is to illustrate the use of a static data member to maintain a single, shared count of all objects created for a class, and a static member function to access and display this shared count without needing an object instance.

Algorithm:

Start the program.

Define a class, for example, Test.

Declare a non-static integer data member (id) and a static integer data member (count) inside the class.

Define the static data member count outside the class, initializing it to 0. This allocates memory for the shared variable.

Define a public non-static member function (set_id()) that assigns a unique ID to each object by incrementing the static count and then assigning the new value to the non-static id member.

Define a static member function (show_count()) that can access and display the value of the static count.

Create multiple objects (t1, t2, t3) of the class in the main() function.

Call the non-static member function set_id() for each object to increment the shared count and set their unique ID.

Call the static member function show_count() using the class name and the scope resolution operator (::) to display the total number of objects created, which is the shared count value. This can be done with or without creating an object.

Stop the program.

Program :

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

class Test {
private:
    int id;
    static int count; // Declaration of a static data member
public:
    void set_id() {
        count++; // Increments the single shared copy
        id = count;
    }
    void show_id() {
        cout << "Object ID: " << id << endl;
    }
    static void show_count() { // Declaration of a static member function
        // Can only access static members
        cout << "Total objects created: " << count << endl;
    }
};

// Definition and initialization of the static data member outside the class
int Test::count = 0;

int main() {
    Test t1, t2; // Objects created

    t1.set_id();
    t2.set_id();

    // Calling static member function using the class name (recommended)
    Test::show_count();
```

```
Test t3; // Another object created
    t3.set_id();

    t1.show_id();
    t2.show_id();
    t3.show_id();

    // Calling static member function using an object (valid but not recommended)
    t3.show_count();

    return 0;
}
```

Output:

```
Total objects created: 2
Object ID: 1
Object ID: 2
Object ID: 3
Total objects created: 3
```

Result:

Thus, a C++ program was written and executed successfully to illustrate the use of static data members and static member functions. The program demonstrated that a static data member (count) is shared by all objects of the class (only one copy exists), and its value is updated across all instances. A static member function (show_count()) was used to access this shared data member without the need for a specific object instance, highlighting its class-level association.

# Experiment-7:Illustrate the use of array of objects

Aim:

The aim is to demonstrate the use of an array of objects in C++ by creating a class Employee to store details (ID and name) of multiple employees and then inputting and displaying these details using an array of Employee objects.

Algorithm:

Define a Class: Create a class (e.g., Student) with relevant attributes (e.g., name as string, age as integer) and a constructor to initialize these attributes.

Declare an Array of Objects: In the main function or another appropriate scope, declare an array of objects of the defined class. The size of the array can be fixed or determined dynamically.

Populate the Array: Iterate through the array, creating instances of the class and assigning them to each element of the array.

This typically involves taking input for each object's attributes.

Access and Manipulate Objects: Iterate through the array again to access the attributes of individual objects and perform operations on them (e.g., display information).

Program:

```
#include <iostream>
#include <string>
#include <vector> // Using vector for dynamic array

class Student {
public:
    std::string name;
    int age;

    // Constructor
    Student(std::string n, int a) : name(n), age(a) {}

    // Method to display student information
    void displayStudentInfo() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    int numStudents;
    std::cout << "Enter the number of students: ";
    std::cin >> numStudents;

    // Create a vector (dynamic array) of Student objects
    std::vector<Student> students;

    // Populate the array of objects
    for (int i = 0; i < numStudents; ++i) {
        std::string studentName;
        int studentAge;

        std::cout << "Enter name for student " << i + 1 << ": ";
        std::cin >> studentName;
        std::cout << "Enter age for student " << i + 1 << ": ";
        std::cin >> studentAge;

        students.emplace_back(studentName, studentAge); // Add new Student object
    }

    std::cout << "\n--- Student Information ---" << std::endl;
    // Access and display information for each student
    for (const auto& student : students) {
        student.displayStudentInfo();
    }

    return 0;
}
```

Output:

```
Enter the number of students: 2
Enter name for student 1: Alice
Enter age for student 1: 20
Enter name for student 2: Bob
Enter age for student 2: 22

--- Student Information ---
Name: Alice, Age: 20
Name: Bob, Age: 22
```

Result:

The program successfully demonstrates the use of an array of objects.

# Experiment-8: Ilustrate the memory management operator

Aim:

The aim is to demonstrate dynamic memory management, where memory is allocated from the heap at runtime (when the exact size needed is unknown at compile time) and subsequently deallocated when no longer needed, to prevent memory leaks and optimize memory use.

Algorithm:

Declare a pointer: Create a pointer variable to store the address of the dynamically allocated memory block.

Allocate memory: Use the new operator to request a block of memory of the required type and size from the system's heap and assign the address of the allocated memory to the pointer.

Handle potential failure: Check if the allocation was successful (e.g., in some contexts, new might return a null pointer if memory is full, though in C++ it usually throws an exception).

Use the memory: Access and manipulate the allocated memory using the pointer (e.g., store data, access object members).

Deallocate memory: Use the delete operator to return the previously allocated memory block to the free store (heap) when it is no longer required.

Avoid dangling pointers: Set the pointer to nullptr after deallocation to prevent accidental use of a dangling pointer.

Program:

```
#include <iostream>

int main() {
    // 1. Declare a pointer
    int* ptr = nullptr;

    // 2. Allocate memory for a single integer
    ptr = new int;

    // Check if allocation failed (optional, as new usually throws exception)
    if (ptr == nullptr) {
        std::cout << "Memory allocation failed!" << std::endl;
        return 1;
    }

    // 4. Use the memory (assign a value)
    *ptr = 100;
    std::cout << "Value stored in dynamically allocated memory: " << *ptr << std::endl;

    // Allocate memory for an array of integers
    int* arr_ptr = new int[5];
    for (int i = 0; i < 5; ++i) {
        arr_ptr[i] = i * 10;
    }
    std::cout << "Values in the dynamically allocated array: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << arr_ptr[i] << " ";
    }
    std::cout << std::endl;

    // 5. Deallocate memory
    delete ptr;
    // 6. Avoid dangling pointers
    ptr = nullptr;

    // Deallocate the array memory using delete[]
    delete[] arr_ptr;
    // Avoid dangling pointers
    arr_ptr = nullptr;
```

```
std::cout << "Memory deallocated successfully." << std::endl;

    return 0;
}
```

Output:

```
Value stored in dynamically allocated memory: 100
Values in the dynamically allocated array: 0 10 20 30 40
Memory deallocated successfully.
```

Result:

The program successfully demonstrates dynamic memory management in C++. It uses the new operator to allocate memory blocks from the heap for both a single integer and an array of integers during runtime. The program then stores and accesses data in these memory locations. Finally, the delete and delete[] operators are used to explicitly deallocate the memory, returning it to the system for reuse and preventing memory leaks.

# Experiment-9:Illustrate the use of friend class and friend function

Aim:

To implement and demonstrate the use of Unary Operator Overloading in C++

Algorithm:

1. Start the program.

2. Create a class with a private data member.

3. Define a constructor to initialize the data.

4. Overload the unary minus (–) operator using a member function.

5. Inside the overloaded function, negate the value of the data member.

6. Display the values before and after applying the operator.

7. End the program.

Program:

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

class A {
private:
    int x;
public:
    A() { x = 10; }
    friend class B;           // Friend class
    friend void show(A obj);  // Friend function
};

class B {
public:
    void display(A obj) {
        cout << "Friend Class Access: x = " << obj.x << endl;
    }
};

void show(A obj) {
```

```
cout << "Friend Function Access: x = " << obj.x << endl;
}

int main() {
    A objA;
    B objB;

    objB.display(objA);   // Friend class accessing private data
    show(objA);           // Friend function accessing private data

    return 0;
}
```

Output:

```
Friend Class Access: x = 10
Friend Function Access: x = 10
```

Result:

The program successfully shows that both friend class and friend function can access the private members of another class directly.

# Experiment-10:Implement the use of unary operator overloading

Aim:

To implement and demonstrate the use of Unary Operator Overloading in C++.

Algorithm:

1. Start the program.

2. Create a class with a private data member.

3. Define a constructor to initialize the data.

4. Overload the unary minus (–) operator using a member function.

5. Inside the overloaded function, negate the value of the data member.

6. Display the values before and after applying the operator.

7. End the program.

Program:

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

class Number {
private:
    int x;
public:
    Number(int a) {
        x = a;
    }

    // Overload unary minus operator
    void operator-() {
        x = -x;
    }
```

```
// Overload unary minus operator
    void operator-() {
        x = -x;
    }

    void display() {
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    Number n1(10);

    cout << "Before overloading:" << endl;
    n1.display();

    -n1;  // calls operator-()

    cout << "After overloading:" << endl;
    n1.display();

    return 0;
}
```

Output:

```
Before overloading:
Value of x: 10
After overloading:
Value of x: -10
```

Result:

The program successfully demonstrates unary operator overloading.
The unary minus (–) operator is overloaded to change the sign of the private data member x.