OOP

What is OOP?

- OOP stands for Object-Oriented Programming.
- It is a programming approach in which we design program using classes and objects.
- It is a way of structuring and designing programs to represent real-world entities or concepts.
- The main aim of OOP is to bind together the data and the function that operate on them. So that no other part of data can access this data except this function.

Primitive Data Types:

- Primitive data types are the fundamental or basic data types provided by a programming language.
- Primitive data types have a fixed size in memory and are typically directly stored in variables, making them efficient in terms of memory usage and performance.

Exampls:

Integer, Float, Boolean etc

Non-Primitive Data Types:

- Non-primitive data types are created by the programmer using primitive data types or other non-primitive types.
- They are often referred to as composite or reference data types because they contain references (or pointers) to the actual data stored elsewhere in memory.
- They can be dynamically allocated and deallocated from memory and offer more flexibility in terms of defining complex data structures and behaviors.

Examples:

Arrays, Strings, Structures and Classes etc.

In summary, primitive data types are the basic building blocks of a programming language and have fixed sizes, while non-primitive data types are constructed using primitive types or other non-primitive types and provide more complex and flexible data structures.

Structure:

- Structure is a user defined data type that allows you to group together related data elements of different types under a single name.

```
struct Person {
  char name[50];
  int age;
  float height;
};
```

Class:

- A class is a blueprint or a template for creating objects.
- It defines the properties (attributes) and behaviors (methods) that objects of that class can have.

Constructor:

- A constructor is a member function that is automatically called when an object of a class is created.
- Its purpose is to initialize the object and prepare it for use.
- Constructors have the same name as the class and do not have a return type (not even void). They can have parameters, allowing you to pass values to initialize the object's data members.

Destructor:

- A destructor, on the other hand, is a member function that is automatically called when an object is about to be destroyed or deallocated.
- Its purpose is to release any resources that the object may have acquired during its lifetime, such as dynamically allocated memory or file handles.

Encapsulation:

- Encapsulation is one of the fundamental principles of object-oriented programming (OOP).
- It refers to the bundling of data (attributes) and methods (behaviors) together within a class, and controlling access to them through a well-defined interface.
- Encapsulation ensures that the internal workings and implementation details of an object are hidden from the outside world, promoting information hiding and data abstraction.

Abstraction:

- Abstraction is also one of the fundamental principles of object-oriented programming (OOP).
- It is like creating a blueprint or a model that captures the essential features and behavior of an object, without concerning with the internal implementation details.
- only show the necessary information to user not the internal logic.
- Abstraction can be accomplish using classes or header files.

Example:

```
public abstract class Car {
   public abstract void start();
   public abstract void accelerate();
   public abstract void brake();
   public abstract void stop();
}
```

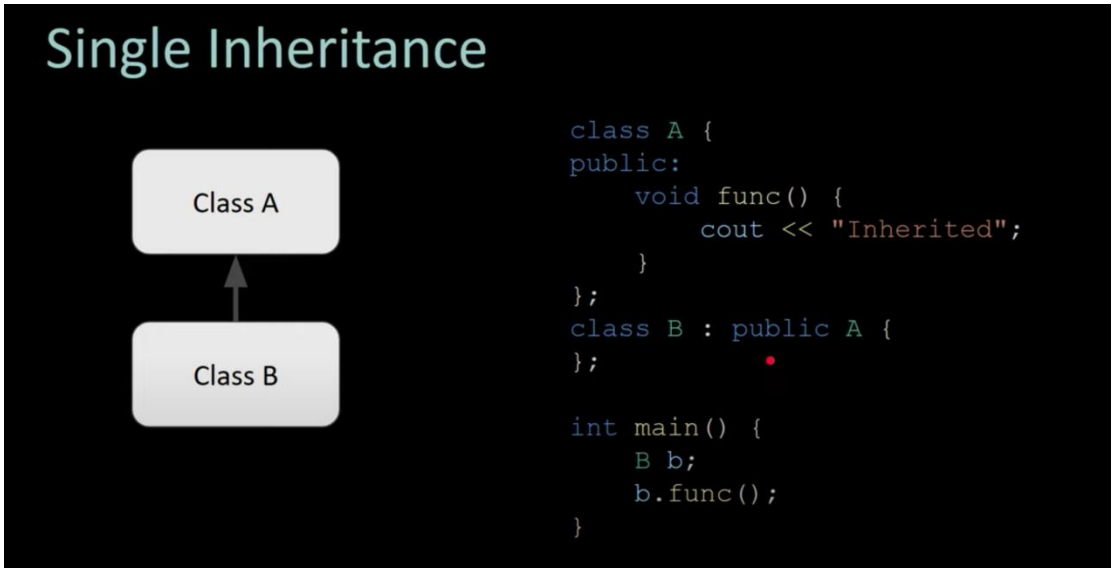
In the above example, we have an abstract class called "Car" that defines the essential behaviors (methods) a car should have, such as starting, accelerating, braking, and stopping. However, the implementation details of these methods are left to the concrete classes that inherit from this abstract class.

Inheritance: (Reuseability)

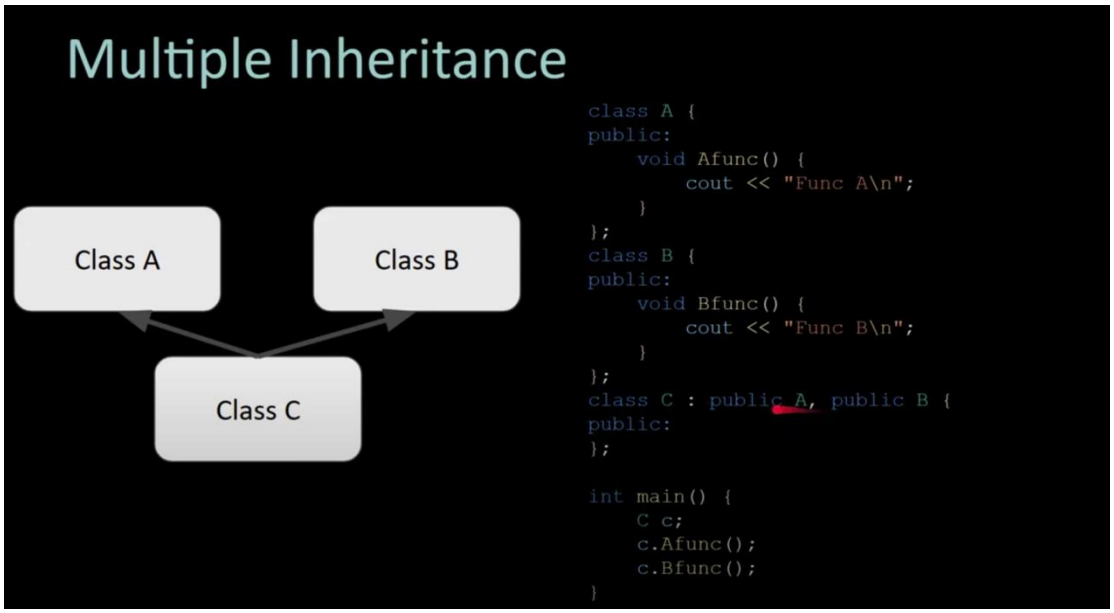
- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties (attributes) and behaviors (methods) from another class.
- The class that inherits is called a derived class or subclass, and the class from which it inherits is called a base class or superclass.

Types:

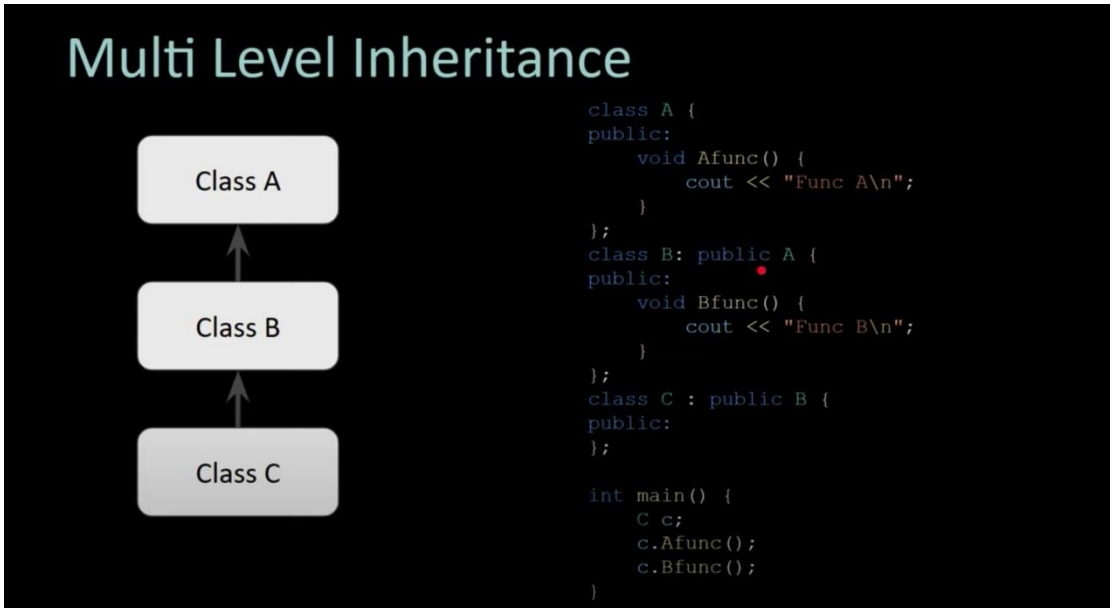
Single Inheritance: In single inheritance, a derived class inherits from a single base class.



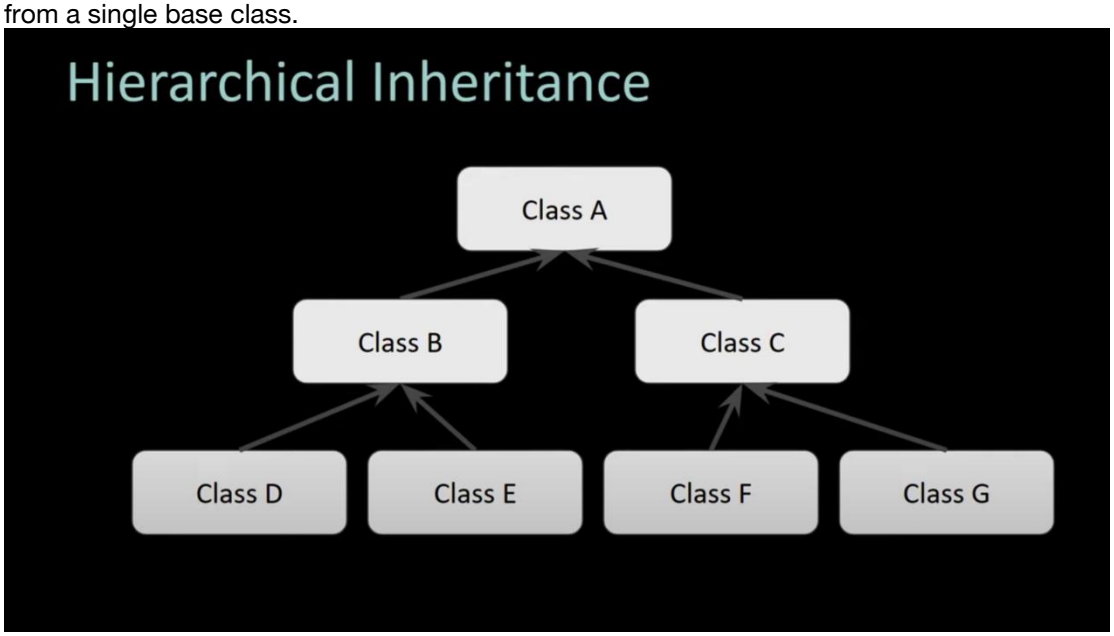
Multiple Inheritance: Multiple inheritance allows a derived class to inherit from multiple base classes.

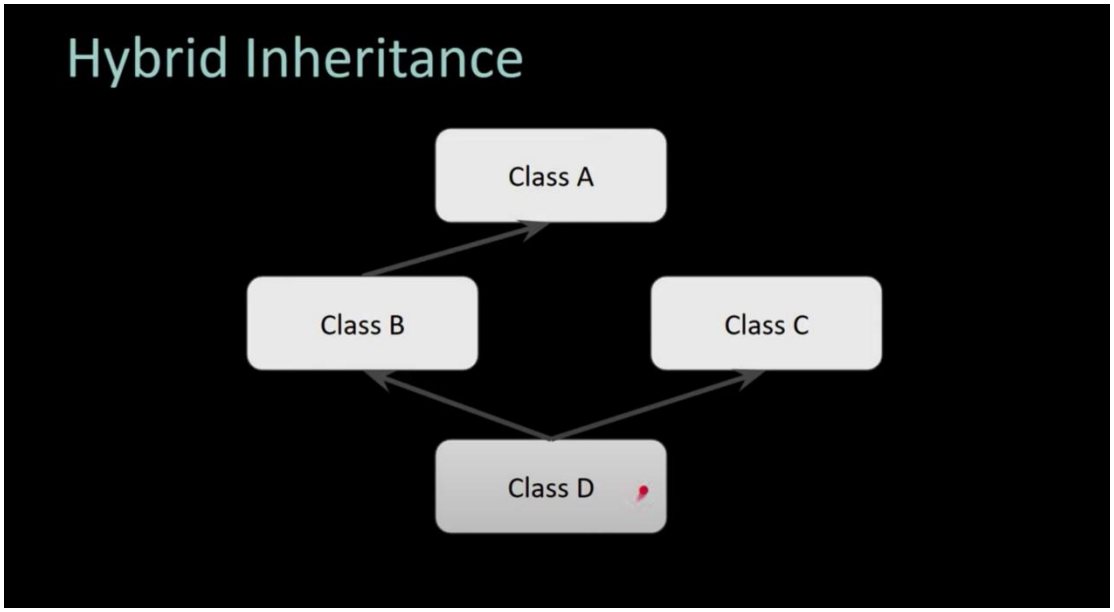


Multilevel Inheritance: Multilevel inheritance involves a chain of inheritance, where a derived class inherits from a base class, and then another class inherits from that derived class. This forms a hierarchical relationship, with each level building upon the previous one.



Hierarchical Inheritance: Hierarchical inheritance is when multiple derived classes inherit from a single base class.





Access modifiers:
access modifiers control the accessibility of class members (attributes and methods) from other parts of the program.

There are typically three main access modifiers:

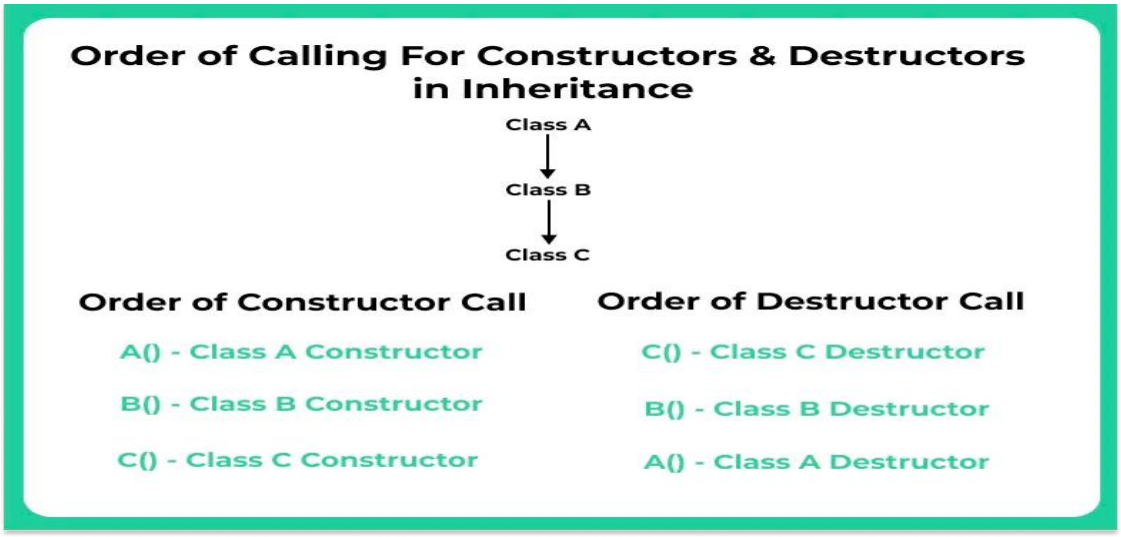
Public: Public members are accessible from anywhere in the program.

Protected: Protected members are accessible within the class and its derived classes (subclasses), but not from external code.

Private: Private members are only accessible within the class itself.

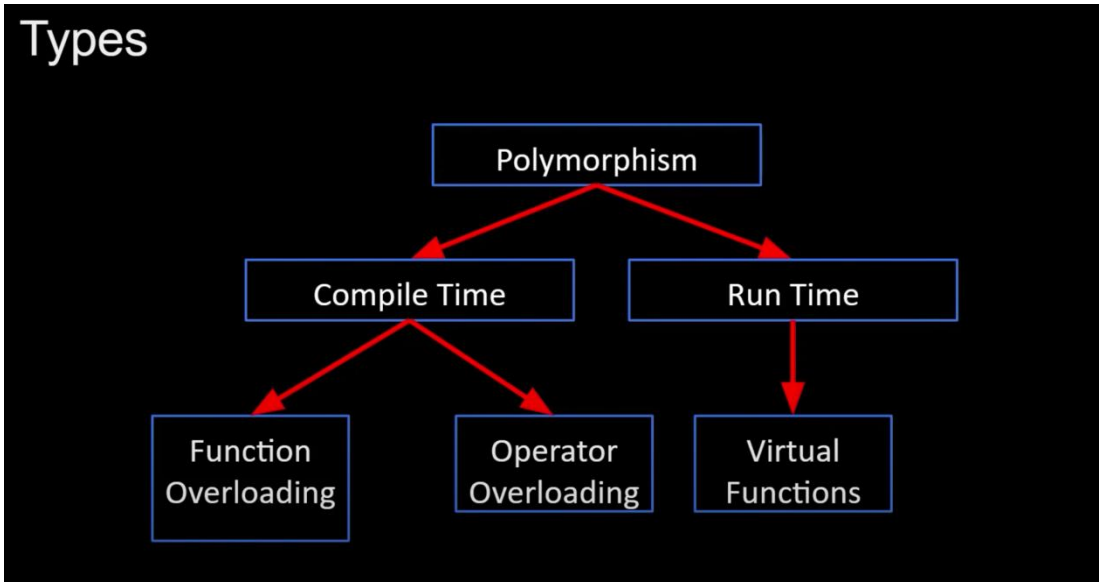
Note: If we don't specify any access modifier, by default the access modifier for the memeber will be private.

Constructor and Destructor in Inheritance:



Polymorphism:

- An entity have different behavior in different scnaerio.



Method Overloading:
Method overloading refers to the ability to define multiple methods with the same name in a class, but with different parameters. These methods can have the same or different return types.

Key points about method overloading:

- Method overloading occurs within a single class.
- Overloaded methods must have the same name but different parameter lists.
- Overloading is determined at compile-time (static polymorphism) based on the arguments used.

Example:

```
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Method Overriding:
Method overriding occurs when a derived class provides its own implementation for a method that is already defined in its base class.

Key points about method overriding:

- Method overriding occurs between a base class and a derived class.
- The overridden method must have the same name, return type, and parameters as the base class method.

- Overriding is determined at runtime (dynamic polymorphism) based on the actual object type.

Example:

```
class Animal {
   void makeSound() {
      System.out.println("Animal is making a sound.");
   }
}

class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.println("Dog is barking.");
   }
}
```

Virtual Function:

- A virtual function is a member function in a base class that is declared with the virtual keyword and is intended to be overridden by derived classes.
- It allows dynamic polymorphism, which means that the appropriate derived class implementation of the virtual function is called based on the actual object type at runtime.

Static Variable in Function:

- When a variable is declared as static, space for it gets allocated for the lifetime of the program.
- Even if the function is called multiple times, the space for it is allocated once.

Example:

```
#include <iostream>

void incrementAndPrint() {
   static int count = 0; // Declaration and initialization of static variable

   count++; // Increment the static variable
   std::cout << "Count: " << count << std::endl;
}

int main() {
   incrementAndPrint(); // Output: Count: 1
   incrementAndPrint(); // Output: Count: 2
   incrementAndPrint(); // Output: Count: 3

   return 0;
}
```

Static Variable in Class:

- Declared inside the class body.
- They must be initilize outside the class.
- Static variable does not belong to any object, but the whole class.
- There will be only one copy in memory for static vaiable for the whole class.

Example:

```
#include <iostream>

class MyClass {
public:
```

```
    static int count; // Declaration of static variable

    MyClass() {
        count++; // Increment the static variable in the constructor
    }
};

// Definition and initialization of the static variable
int MyClass::count = 0;

int main() {
    MyClass obj1;
    MyClass obj2;
    MyClass obj3;

    std::cout << "Total objects created: " << MyClass::count << std::endl; //Output is 3

    return 0;
}
```

Static Member Function:

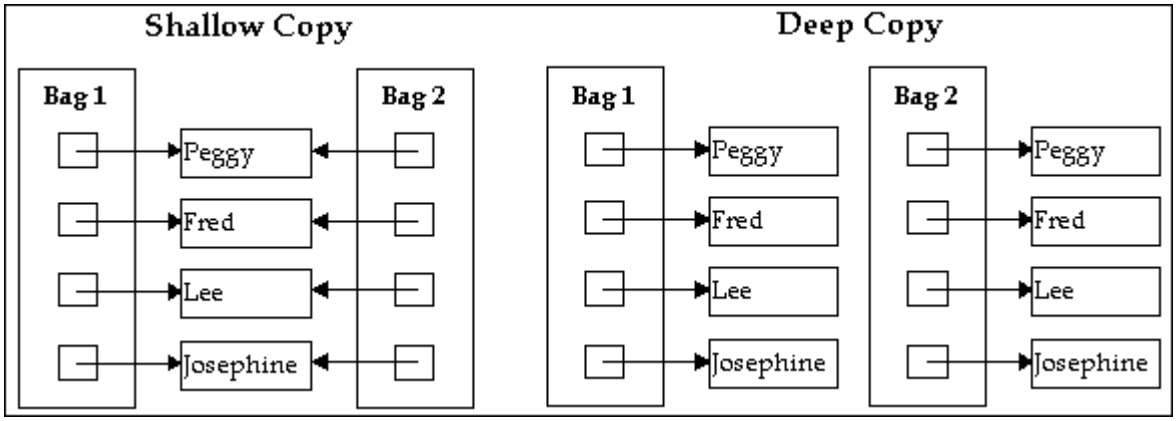
- Static member function in a class is a function that belongs to the class itself rather than to any particular instance or object of the class.
- It is declared with the **static** keyword in the class definition and can be invoked using the class name, without the need to create an object.

Example:

```
#include <iostream>

class MathUtils {
public:
    static int add(int a, int b) {
        return a + b;
    }

    static int multiply(int a, int b) {
        return a * b;
    }
};

int main() {
    int sum = MathUtils::add(3, 4);
    int product = MathUtils::multiply(5, 6);

    std::cout << "Sum: " << sum << std::endl; // Output is 7
    std::cout << "Product: " << product << std::endl; // Output is 30

    return 0;
}
```

Shallow Copy vs Deep Copy:

Shallow copy and deep copy are terms used in the context of object-oriented programming to describe different ways of copying the contents of an object into another object.



Shallow Copy:

- Shallow copy creates a new object and copies the values of the member variables from the source object to the destination object.
- If the object contains pointers to dynamically allocated memory, a shallow copy simply copies the memory addresses without creating new memory blocks. Both the source and destination objects then point to the same memory locations.
- As a result, changes made to the shared memory through one object affect the other object as well since they both refer to the same memory locations.

Example:

```
#include <iostream>

class Employee {
public:
    int* empId;

    Employee(int id) {
        empId = new int(id);
    }

    // Shallow copy constructor
    Employee(const Employee& other) {
        empId = other.empId;
    }

    ~Employee() {
```

```
        delete empId;
    }
};

int main() {
    Employee employee1(123);
    Employee employee2 = employee1;

    // Modify empId of employee2 (affects employee1 as well)
    *(employee2.empId) = 456;

    std::cout << "Employee 1 ID: " << *(employee1.empId) << std::endl; // Output is 456
    std::cout << "Employee 2 ID: " << *(employee2.empId) << std::endl; // Output is 456

    return 0;
}
```

Deep Copy:

- Deep copy also creates a new object and copies the values of the member variables from the source object to the destination object.
- However, in a deep copy, if the object contains pointers to dynamically allocated memory, new memory blocks are allocated in the destination object, and the data is copied into those new blocks.
- This way, the source and destination objects have separate memory blocks, and changes made to one object do not affect the other object.

Example:

```
#include <iostream>

class Employee {
public:
    int* empId;

    Employee(int id) {
        empId = new int(id);
    }

    // Deep copy constructor
    Employee(const Employee& other) {
        empId = new int(*(other.empId));
    }

    ~Employee() {
        delete empId;
    }
};

int main() {
    Employee employee1(123);
    Employee employee2 = employee1;

    // Modify empId of employee2 (does not affect employee1)
    *(employee2.empId) = 456;

    std::cout << "Employee 1 ID: " << *(employee1.empId) << std::endl; // Output is 123
    std::cout << "Employee 2 ID: " << *(employee2.empId) << std::endl; // Output is 456
```

```
    return 0;
}
```

Friend Class:
Friend class is a class that is granted access to the private and protected members of another class.

Example:

```
#include <iostream>

class MyClass {
private:
    int privateData;

public:
    MyClass(int data) : privateData(data) {}

    friend class FriendClass;
};

class FriendClass {
public:
    void accessPrivateData(MyClass& obj) {
        std::cout << "Accessing private data of MyClass: " << obj.privateData << std::endl;
    }
};

int main() {
    MyClass obj(42);
    FriendClass friendObj;
    friendObj.accessPrivateData(obj); // Output is 42

    return 0;
}
```