C++ Object Oriented Programming

Outline

- C++ Class and Objects
- Inheritance in c++
- Polymorphism in c++
- Encapsulation in c++
- Function Overloading in c++
- Exception Handling in c++

Object oriented programming

- programming paradigm that uses classes and objects to structure code and data.
- object-oriented programming is about creating objects that contain both data and functions.

Procedural programming vs oop

- Procedural programming and object-oriented programming (OOP) are two different programming paradigms.
- Procedural programming is a programming paradigm *that structures code around procedures, functions, or subroutines*.
- Procedural programming focuses on the sequence of steps taken to solve a problem and emphasizes the use of functions to modularize code.

Cont..

- OOP is a programming paradigm that structures code around objects and classes.
- OOP is a programming paradigm focuses on the data and behavior of objects and emphasizes the use of inheritance, polymorphism, and encapsulation.

Advantages of OOP

- **Encapsulation:** OOP allows you to encapsulate data and behavior into objects, which helps to protect your code and prevent unintended changes to the data.
- **Abstraction:** OOP allows you to abstract away the implementation details of a class and focus on its interface, which makes it easier to use and understand.

Cont..

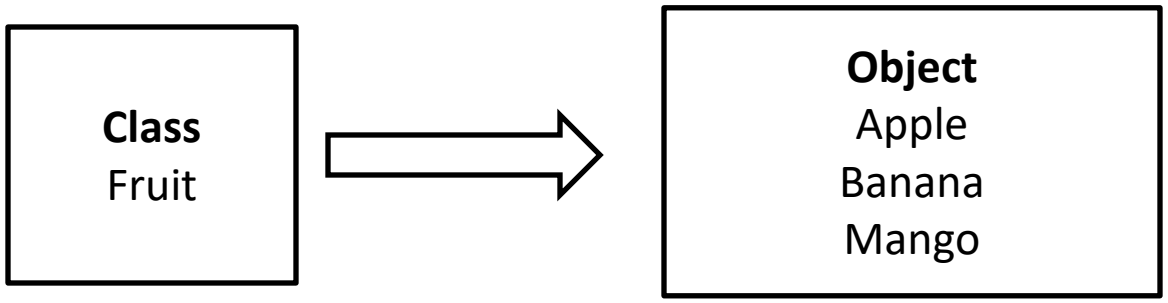
- **Inheritance:** OOP allows you to create new classes that inherit properties and methods from existing classes, which can save time and reduce code duplication.
- **Polymorphism:** OOP allows you to write code that can work with objects of different types, which makes your code more flexible and reusable.
- **Modularity:** OOP allows you to modularize your code into smaller, more manageable pieces, which makes it easier to maintain and update.

Class and object

- a class is a blueprint or template for creating objects.
- It defines the properties and methods that an object will have.
- class is a user-defined data type that encapsulates data and functions that operate on that data.
- A class definition typically includes *member variables, which are the data components of the class*, and *member functions, which are the functions that operate on that data.*
- The member variables and functions are encapsulated within the class,

Cont..

- object is an instance of a class.
- It is created from the blueprint or template defined by the class.
- it has its own set of values for the member variables defined in the class.



Create a class

Step 1: Define the class: Use the "class" keyword followed by the name of the class to define the class.

Syntax:

class className {

Access modifiers: // member variables and functions go here };

Example:

```
class people{
public:
   // member variables and functions go here
};
```

Cont..

Step 2: Define member variables: Inside the class definition, define any member variables that the class will have.

- Member variables are the data components of the class and are accessed using the dot notation. For example:

```
class people {
public:
   string name;
   int age;
};
```

cont..

Step 3: Define member functions: Inside the class definition, define any member functions that the class will have.

- Member functions are the functions that operate on the member variables of the class and are accessed using the dot notation.

For example:

```
class people {
public:
    string name;
    int age;
    void printInfo() {
  cout << "My name is" << name << " and age is" << age << endl;
    }
};
```

object.

Step 5: Create objects: Once you have defined a class, you can create objects of that class using the following syntax:

className objectName;

Example:

people person1;

This creates an object called “person1" of the “people" class.

Access member variables and functions

Step 5: Access member variables and functions:

- Once you have created an object, you can access its member variables and functions using the dot notation.
- For example:

person1.name = “Tola Gamada";

person1.age = 23;

person1.printInfo();

- This sets the values of the “name" and “age" member variables of "myObject" and then calls its "printInfo" member function.

Example

```
#include <iostream>
#include <string>
using namespace std;
class Person {
public:
    string name;
    int age;
    void printInfo() {
        cout << "My name is " << name << " and I am " << age << " years old." << endl;
    }
};
int main() {
    Person myPerson;
    myPerson.name = "Chaltu";
    myPerson.age = 30;
    myPerson.printInfo();
    return 0;
}
```

Description of example

- In the above example, we define a class called "Person" that has two member variables, "name" and "age," and one member function, "printInfo."
- We then create an object called "myPerson" of the "Person" class, set its "name" and "age" member variables to "Chaltu" and 30, respectively, and call its "printInfo" member function to print out its name and age.

When you run this program, it will output the following:

My name is Chaltu and I am 30 years old.

Example 2

```
#include <iostream>
using namespace std;
class Rectangle {
public:
   double width;
   double height;
   void setDimensions(double w, double h) {
      width = w;
      height = h;
   }
   double getArea() {
      return width * height;
   }
   double getPerimeter() {
      return 2 * (width + height);
   }
};
int main() {
   Rectangle myRect;
   myRect.setDimensions(5.0, 10.0);
   cout << "Area: " << myRect.getArea() << endl;
   cout << "Perimeter: " << myRect.getPerimeter() << endl;
   return 0;
}
```

Description of example 2

- In this example, we define a class called "Rectangle" that has two member variables, "width" and "height," and three member functions, "setDimensions," "getArea," and "getPerimeter."
- We then create an object called "myRect" of the "Rectangle" class, set its dimensions to 5.0 and 10.0 using the "setDimensions" member function, and call its "getArea" and "getPerimeter" member functions to calculate its area and perimeter.
- When you run this program, it will output the following:

Area: 50

Perimeter: 30

Access specifier

- Access specifiers in C++ are keywords used to specify the access rules for the members of a class.
- The three access specifiers :
- Public: Members declared as public are accessible from anywhere in the program.
- Private: Members declared as private are only accessible within the class in which they are declared.
- Protected: Members declared as protected are accessible within the class in which they are declared and also in the derived classes.

```
class Example {
public:
   int publicVar;   // Public member variable
   void publicFunc() {   // Public member function
      // Function code goes here
   }
private:
   int privateVar;  // Private member variable
   void privateFunc() {  // Private member function
      // Function code goes here
   }
protected:
   int protectedVar;  // Protected member variable
   void protectedFunc() {  // Protected member function
      // Function code goes here
   }
};
```

- In this example, publicVar and publicFunc() are declared as public.
- privateVar and privateFunc() are declared as private, and
- protectedVar and protectedFunc() are declared as protected.

Constructor

- In C++, a constructor is a special member function that is called when an object of a class is created.
- The constructor is used to initialize the object's properties and perform any other necessary setup.
- The constructor should have the same name as the class and no return type.
- It can take zero or more parameters, which are used to initialize the member variables.
- Syntax:

```
Classname(list of parameters){
//statements
}
```

Steps to have constructor

1. Define a constructor for the class. The constructor should have the same name as the class and no return type.

Example:

```
class Person {
public:
   string name;
   int age;
   Person(string n, int a) {
      name = n;
      age = a;
   }
};
```

Cont..

2. Use the constructor to create objects of the class. When you create an object, the constructor will be called automatically to initialize its member variables.

```
int main() {
    Person p(“Tola", 25);
    return 0;
}
```

Types constructor:

1. **Default constructor:** A default constructor is a constructor that takes no arguments.
   - If you don't define a constructor for your class, the compiler will provide a default constructor that does nothing.

```
class MyClass {
public:
   MyClass() {
      // default constructor
   }
};
```

Cont..

2. Parameterized constructor: A parameterized constructor is a constructor that takes one or more arguments.

- It is used to initialize the object's properties with specific values.

```
class Person {
public:
   string name;
   int age;
   Person(string n, int a) {
      name = n;
      age = a;
   }
};
```

Inheritance

- Inheritance refers to the ability of a class to inherit properties and behavior from another class.
- The class that is being inherited from is called the base class or parent class, while the class that is inheriting is called the derived class or child class.
- Inheritance allows for code reuse and can simplify code by allowing common functionality to be defined in a base class and then inherited by multiple derived classes.
- In C++, inheritance is implemented using the keyword "class" followed by a colon and the name of the base class.
- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

Advantage of inheritance

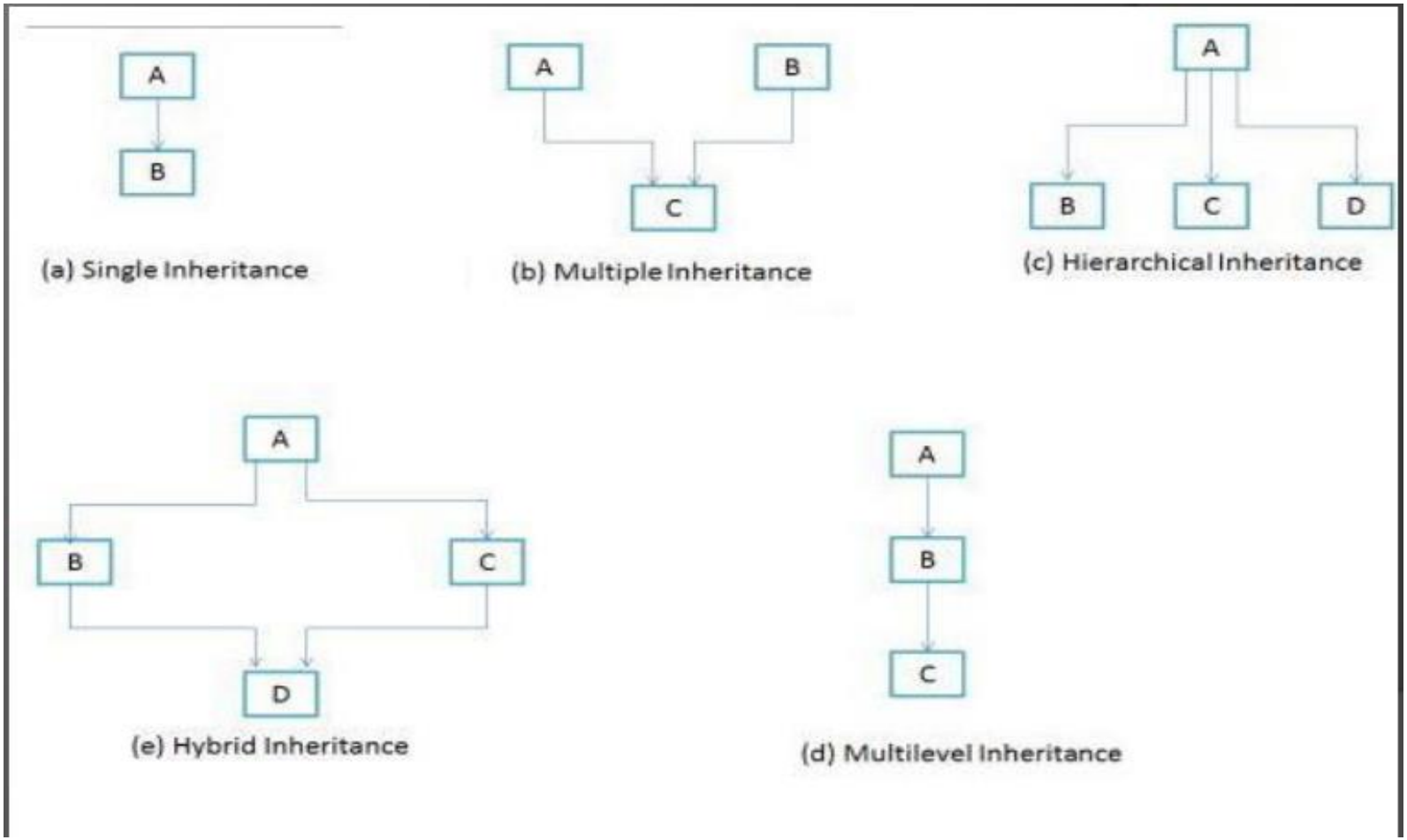
- **Code reusability:-** Inheritance automates the process of reusing the code of the super classes in the subclasses.
  - With inheritance, an object can inherit its more general properties from its parent object, and that saves the redundancy in programming.
- **Code maintenance:-** Organizing code into hierarchical classes makes its maintenance and management easier.
- **Implementing OOP:-** Inheritance helps to implement the basic OOP philosophy to adapt computing to the problem and not the other way around, because entities (objects) in the real world are often organized into a hierarchy.

Types of inheritance

- C++ supports five types of inheritance:

1. **Single inheritance:** A derived class inherits from a single base class. This is the most common type of inheritance.
2. **Multiple inheritance:** A derived class inherits from multiple base classes. This allows the derived class to combine the features of multiple classes.
3. **Multilevel inheritance:** A derived class inherits from a base class, which itself inherits from another base class.
4. Hierarchical inheritance: Multiple derived classes inherit from a single base class.
5. **Hybrid inheritance:** This is a combination of multiple and multilevel inheritance. It allows a class to inherit from multiple classes using both direct and indirect inheritance.

Cont..


A
B
(a) Single Inheritance
A
B
C
(b) Multiple Inheritance
A
B
C
D
(c) Hierarchical Inheritance
A
B
C
D
(e) Hybrid Inheritance
A
B
C
(d) Multilevel Inheritance

Single inheritance

```
#include <iostream>
Using namespace std;
class Animal {
public:
   void eat() {
      cout << "The animal is eating." << endl;
   }
   void sleep() {
      cout << "The animal is sleeping." << endl;
   }
};
class Dog : public Animal {
public:
   void bark() {
      cout << "The dog is barking." << endl;
   }
};
int main() {
   Dog myDog;
   myDog.eat();
   myDog.sleep();
   myDog.bark();
}
```

- In this example, we create an instance of the Dog class called myDog.
- We then call the eat(), sleep(), and bark() functions on myDog.
- Since the Dog class inherits from the Animal class, it has access to the eat() and sleep() functions, and it also has its own unique function called bark().

Hierarchical inheritance

```
#include <iostream>
Using namespace std;
class Animal {
public:
   void eat() {
      cout << "The animal is eating." << endl;
   }
   void sleep() {
      cout << "The animal is sleeping." << endl;
   }
};
class Dog : public Animal {
public:
   void bark() {
      cout << "The dog is barking." << endl;
   }
};
class Cat : public Animal {
public:
   void purr() {
      cout << "The cat is purring." << endl;
   }
};
class Lion : public Animal {
public:
   void roar() {
      cout << "The lion is roaring." << endl;
   }
};
int main() {
   Dog myDog;
   Cat myCat;
   Lion myLion;
   myDog.eat();
   myDog.sleep();
   myDog.bark();
   myCat.eat();
   myCat.sleep();
   myCat.purr();
   myLion.eat();
   myLion.sleep();
   myLion.roar();
   return 0;
}
```

Hybrid inheritance

```
#include <iostream>
Using namespace std;
class Animal {
public:
   void eat() {
      cout << "The animal is eating." << endl;
   }
   void sleep() {
      cout << "The animal is sleeping." << endl;
   }
};
class Mammal : public Animal {
public:
   void giveBirth() {
      cout << "The mammal is giving birth." << endl;
   }
};
class Bird : public Animal {
public:
   void layEggs() {
      cout << "The bird is laying eggs." << endl;
   }
};
class Platypus : public Mammal, public Bird {
public:
   void swim() {
      cout << "The platypus is swimming." << endl;
   }
};
int main() {
   Platypus myPlatypus;
   myPlatypus.eat();
   myPlatypus.sleep();
   myPlatypus.giveBirth();
   myPlatypus.layEggs();
   myPlatypus.swim();
}
```

Multiple inheritance

```
#include <iostream>
using namespace std;
class Mammal {
public:
   Mammal() {
      cout << "Mammals can give direct birth." << endl;
   }
};
class WingedAnimal {
public:
   WingedAnimal() {
      cout << "Winged animal can flap." << endl;
   }
};
class Bat : public Mammal, public WingedAnimal {};
int main() {
   Bat b1;
   return 0;
}
```

Multilevel inheritance

```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Animal is eating." << endl;
    }
};
class Mammal : public Animal {
public:
    void giveBirth() {
        cout << "Mammal is giving birth." << endl;
    }
};
class Dog : public Mammal {
public:
    void bark() {
        cout << "Dog is barking." << endl;
    }
};
int main() {
    Dog myDog;
    myDog.eat(); // inherited from Animal
    myDog.giveBirth(); // inherited from Mammal
    myDog.bark(); // defined in Dog
    return 0;
}
```

Polymorphism

- Polymorphism is an important concept of object-oriented programming.
- It simply means more than one form.
- That is, the same entity (function or operator) behaves differently in different scenarios.
- Polymorphism refers to the ability of a single function or operator to behave differently in different situations.

types of polymorphism

- Compile-time polymorphism (also known as static polymorphism) - This is achieved using function overloading and templates.
- Run-time polymorphism (also known as dynamic polymorphism) - This is achieved using virtual functions and inheritance.

Compile-time polymorphism

- Function overloading allows multiple functions with the same name to be defined in a single scope, as long as their parameter lists are different.
- The appropriate function to call is determined by the compiler based on the number and types of arguments passed to the function.

example

```
#include <iostream>
using namespace std;
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};
int main() {
    Calculator calc;
    int a = 5, b = 10;
    double c = 3.14, d = 2.71;
cout << "int sum: " << calc.add(a, b) << endl; // calls add(int, int)
cout << "double sum: " << calc.add(c, d) << endl; // calls add(double, double)
    return 0;
}
```

- In this example, the Calculator class overloads the add() function with two different versions that take integer and double parameters respectively.
- The appropriate version of the function is called based on the types of arguments passed to it.

Run-time polymorphism

- n C++ inheritance, we can have the same function in the base class as well as its derived classes.
- When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.
- So, different functions are executed depending on the object calling the function.
- This is known as **function overriding** in C++.

Example.

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

Encapsulation

- It is a mechanism that allows the implementation details of a class to be hidden from the outside world, while exposing a public interface that can be used to interact with the class.
- Public members can be accessed from anywhere, including outside the class. They define the public interface of the class.
- Private members are only accessible from within the class. They define the implementation details of the class and are hidden from the outside world.
- Protected members are similar to private members, but can be accessed by derived classes.

Cont..

Encapsulation provides several benefits, including:

- Improved code organization and maintainability
- Reduced coupling between different parts of the code
- Increased security and data integrity
- Easier debugging and error handling

Cont..

```
class BankAccount {
private:
   int accountNumber;
   double balance;
public:
   void deposit(double amount) {
      balance += amount;
   }
   void withdraw(double amount) {
      if (balance >= amount) {
         balance -= amount;
      } else {
         cout << "Insufficient funds" << endl;
      }
   }
   double getBalance() {
      return balance;
   }
};
```

Cont..

- In this example, the BankAccount class has two private data members (accountNumber and balance) and three public member functions (deposit(), withdraw(), and getBalance()).
- The private data members are not accessible from outside the class, so they are encapsulated.
- The public member functions provide a public interface for interacting with the BankAccount class.
- The deposit() and withdraw() functions modify the balance data member, while the getBalance() function returns the current balance.
- By encapsulating the implementation details of the BankAccount class, we can ensure that the data is only modified in a controlled way, which helps to prevent bugs and improve code reliability.