OBJECT PROGRAMMING

- LECTURE 1-

(1ST YEAR OF STUDY)

Skopje, 2023/24

Contents

1.1. Introduction

- **Object-oriented Programming** – As the name suggests uses ***objects*** in programming.
- Object-oriented Programming (OOP) aims to implement real-world entities, like: *inheritance*, *hiding*, *polymorphism*, etc. in programming.
- The main aim of OOP is to bind together the **data** and the **functions** that *operate on them*, so that no other part of the code can access this data except that function.

1.1. Introduction

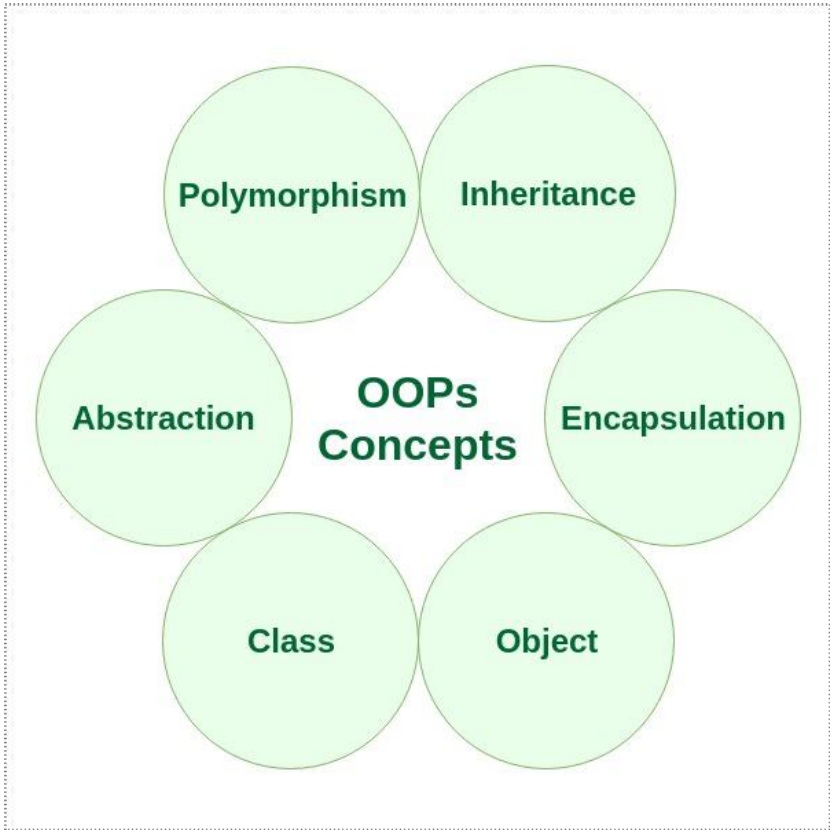
- *Object-oriented Programming* (OOP) is a programming paradigm where the complete software operates as a bunch of objects talking to each other.
- An ***object*** is a collection of *data* and the *methods* which operate on that data.
- The OOP paradigm is mainly useful for relatively big software.

1.1. Introduction

- There are some basic concepts that act as the ***building blocks*** of OOPs, i.e.,
  - Class
  - Objects
  - Encapsulation
  - Abstraction
  - Polymorphism
  - Inheritance
  - Dynamic Binding
  - Message Passing

1.1. Introduction

- **Characteristics of an OOP Language**, which give a clear structure to programs, and allow code to be reused, lowering development costs and providing security.



1.1. Introduction

- C++ is the most used and popular cross-platform programming language.
- C++ improves on many of C's features and provides Object-oriented Programming (OOP) capabilities that increase software productivity, quality and reusability.
- When a programming language becomes as entrenched as C, new requirements demand that the language evolve rather than simply be displaced by a new language.
- C++ was developed by Bjarne Stroustrup, at Bell Laboratories, and was originally called "C with classes".

1.1. Introduction

- The name C++ includes C's increment operator (++) to indicate that C++ is an enhanced version of C.
- C++ is a high-level and Object-oriented Programming language. This language allows developers to write clean and efficient code for large applications and software development, game development, and operating system programming. C++ gives a high level of control over system resources and memory.
- It is an expansion of the C programming language to include Object-oriented Programming (OOP), and is used to develop programs for computers.

1.1. Introduction

Object-Oriented Programming	Structural Programming
Programming that is object-oriented is built on objects having a state and behavior.	A program's logical structure is provided by structural programming, which divides programs into their corresponding functions.
It follows a bottom-to-top approach.	It follows a Top-to-Down approach.
Restricts the open flow of data to authorized parts only providing better data security.	No restriction to the flow of data. Anyone can access the data.
Enhanced code reusability due to the concepts of polymorphism and inheritance.	Code reusability is achieved by using functions and loops.
In this, methods are written globally and code lines are processed one by one i.e., Run sequentially.	In this, the method works dynamically, making calls as per the need of code for a certain time.
Modifying and updating the code is easier.	Modifying the code is difficult as compared to OOPs.
Data is given more importance in OOPs.	Code is given more importance.

1.2. Classes and Objects

- The building block of C++ that leads to Object-oriented programming is a **Class**.
- A Class is a *user-defined data type*, that has its own ***data members*** and ***member functions***, which can be accessed and used by creating an **instance** of that class.
- A *Class* is like a *blueprint* for an *object*.
- Data members are the *data variables* and member functions are the *functions used to manipulate these variables*.
- Together these data members and member functions define the *properties* and *behavior* of the *objects* in a *class*.

1.2. Classes and Objects

- For example: Consider the ***Class of Cars***. There may be many cars with different names (brands), but all of them will share some common properties, like all of them will have 4 wheels, Speed limit, Mileage range, etc. So here, the Car is the ***class***, and wheels, speed limits, and mileage are their ***properties***.
- In the above example of class Car, the data member will be speed limit, mileage, etc. and member functions can apply brakes, increase speed, etc.
- We can say that a ***Class in C++*** is a blueprint representing a group of objects, which shares some common properties and behaviors.

1.2. Classes and Objects

- An **Object** is an identifiable entity with some characteristics and behavior.
- An Object is an *instance* of a Class.
- When a class is *defined*, no memory is allocated, but when it is *instantiated* (i.e., an object is created), memory is allocated.
- Objects take up *space* in memory and have an *associated address*. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code.

1.2. Classes and Objects


```
// C++ Program to show the syntax/working of Objects as a
// part of Object Oriented PProgramming
#include <iostream>
using namespace std;

class person {
    char name[20];
    int id;

public:
    void getdetails() {}
};

int main()
{

    person p1; // p1 is a object
    return 0;
}
```

1.2. Classes and Objects


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

class Student{
  private:
    string name;
    string surname;
    int rollNo;

  public:
    Student(string studentName, string studentSurname, int studentRollNo){
        name = studentName;
        surname = studentSurname;
        rollNo = studentRollNo;
    }

    void getStudentDetails(){
        cout << "The name of the student is " << name << " " << surname << endl;
        cout << "The roll no of the student is " << rollNo << endl;
    }
};

int main() {
    Student student1("Vivek", "Yadav", 20);
    student1.getStudentDetails();

    return 0;
}
```

Output

```
The name of the student is Vivek Yadav
The roll no of the student is 20
```

1.2. Classes and Objects

- The ***structure*** is also a user-defined data type in C++, similar to the class with the following differences:
  - The major difference between a structure and a class is that in a structure, the members are set to ***public*** by default, while in a class, members are ***private*** by default.
  - The other difference is that we use **struct** for declaring structure, and **class** for declaring a class in C++.

1.3. Encapsulation

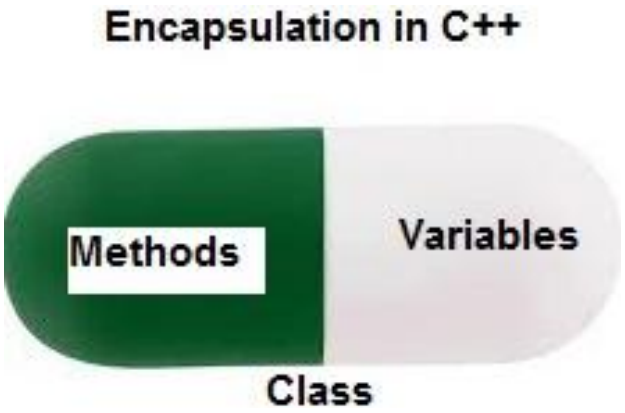
- In normal terms, **Encapsulation** is defined as wrapping up data and information under a single unit.
- In Object-oriented Programming, *Encapsulation* is defined as binding together the *data* and the *functions* that manipulate them.
- Consider a real-life example of encapsulation, in a company, there are different sections: like the accounts section, finance section, sales section, etc.

1.3. Encapsulation

- The **finance section** handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the **sales section** handles all the sales-related activities and keeps records of all the sales.
- Now, there may arise a situation when, for some reason, an official from the *finance section* needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the *sales section*.
- He will first have to contact some other officer in the *sales section* and then request him to give the particular data. This is what *encapsulation* is.

1.3. Encapsulation

- Here the data of the *sales section* and the employees that can manipulate them are wrapped under a single name "sales section".
- Encapsulation also leads to ***data abstraction*** *or* ***data hiding***. Using encapsulation also hides the data. In the above example, the data of any of the sections, like: sales, finance, or accounts, are hidden from any other section.



1.3. Encapsulation

- ***Access specifiers*** are special types of keywords that are used to specify or control the accessibility of entities, like: classes, methods, and so on.
- **Private**, **Public**, and **Protected** are examples of access specifiers, or access modifiers.
- The key components of OOP, *encapsulation* and *data hiding*, are largely achieved because of these access specifiers.

1.4. Abstraction

- **Data abstraction** is one of the most essential and important features of object-oriented programming in C++.
- ***Abstraction*** means displaying only essential information, and hiding the details.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details, or implementation.

1.4. Abstraction

- Consider a real-life example of a man driving a car.
- The man only knows that pressing the accelerator will *increase* the speed of the car, or applying brakes will *stop* the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car, or the implementation of an accelerator, brakes, etc. in the car. This is what *abstraction* is.

1.4. Abstraction

- ***Abstraction using Classes***: We can implement Abstraction in C++ using classes.
  - The class helps us to group *data members* and *member functions* using available access specifiers.
  - A Class can decide which data member will be visible to the outside world, and which is not.

1.4. Abstraction

- ***Abstraction in Header files***: One more type of abstraction in C++ can be header files.
  - For example, consider the pow() method present in math.h header file.
  - Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file, and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

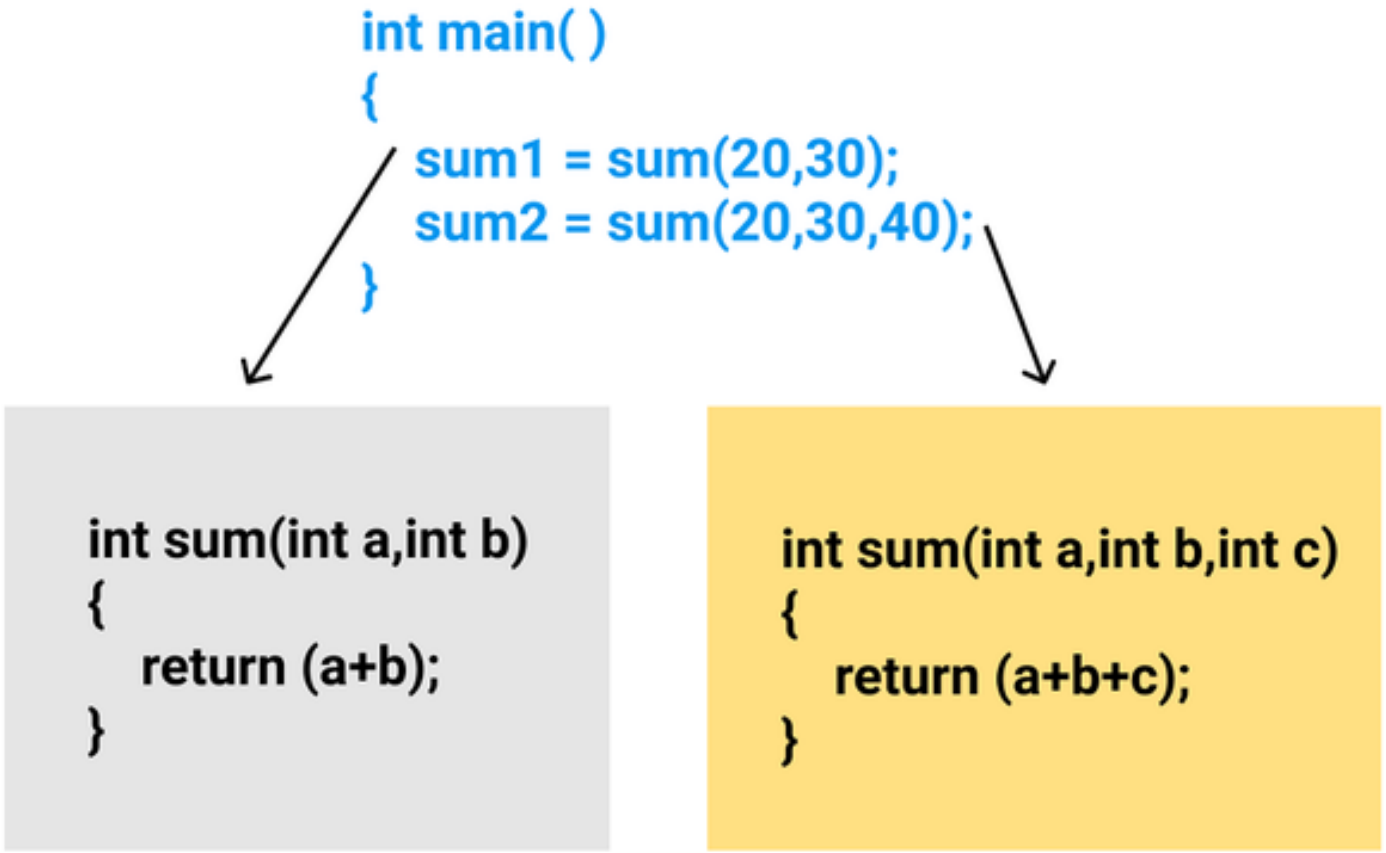
1.5. Polymorphism and Inheritance

- The word **Polymorphism** means having many forms. In simple words, we can define *polymorphism* as the ability of a message to be displayed in more than one form.
- A person at the same time can have *different* characteristics. A man at the same time is a father, a husband, and an employee. So, the same person possesses different behavior in different situations. This is called *polymorphism*.
- An *operation* may exhibit different behaviors in different *instances*. The behavior depends upon the types of data used in the operation. C++ supports ***operator overloading*** and ***function overloading***.

1.5. Polymorphism and Inheritance

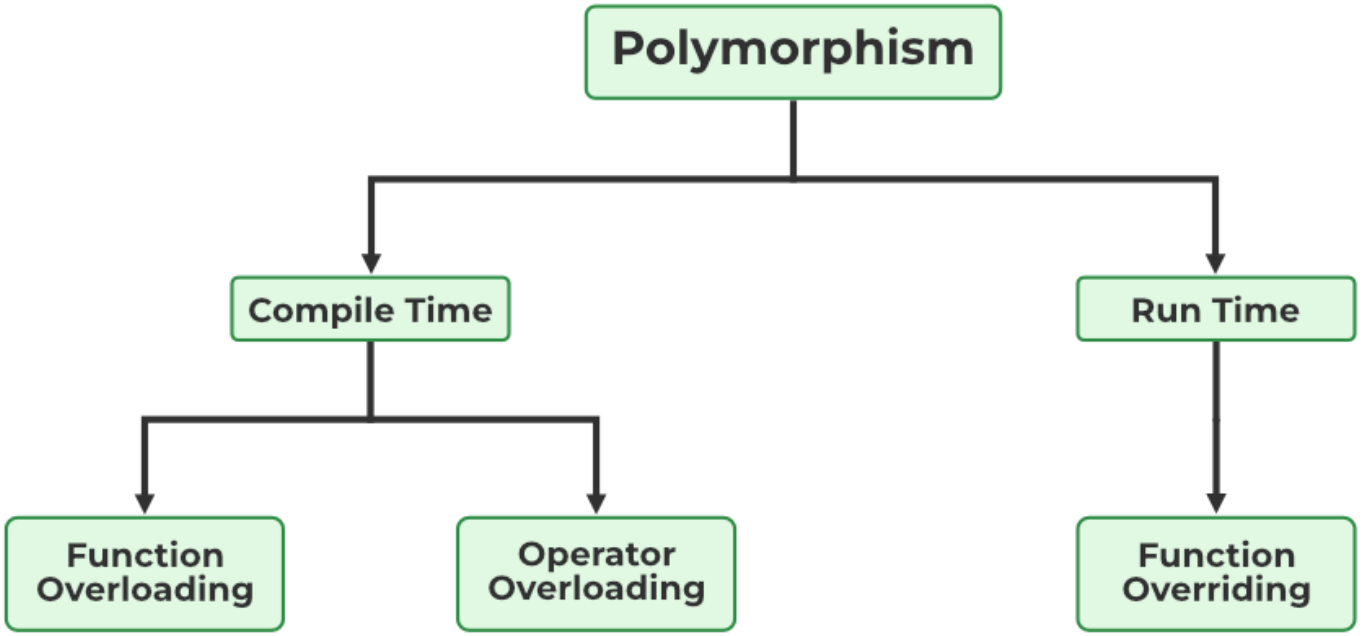
- ***Operator Overloading***: The process of making an operator exhibit different behaviors in different instances is known as *operator overloading*.
- ***Function Overloading***: *Function overloading* is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing ***inheritance***.
- **Example**: Suppose we have to write a function to add some integers, sometimes there are 2 integers, and sometimes there are 3 integers. We can write the Addition Method with the *same name* having different parameters, the concerned *method* will be called according to parameters.

1.5. Polymorphism and Inheritance



1.5. Polymorphism and Inheritance

- *Polymorphism* can be classified into two types, based on the time when the call to the object or function is resolved.
  - Compile Time Polymorphism
  - Run Time Polymorphism



1.5. Polymorphism and Inheritance

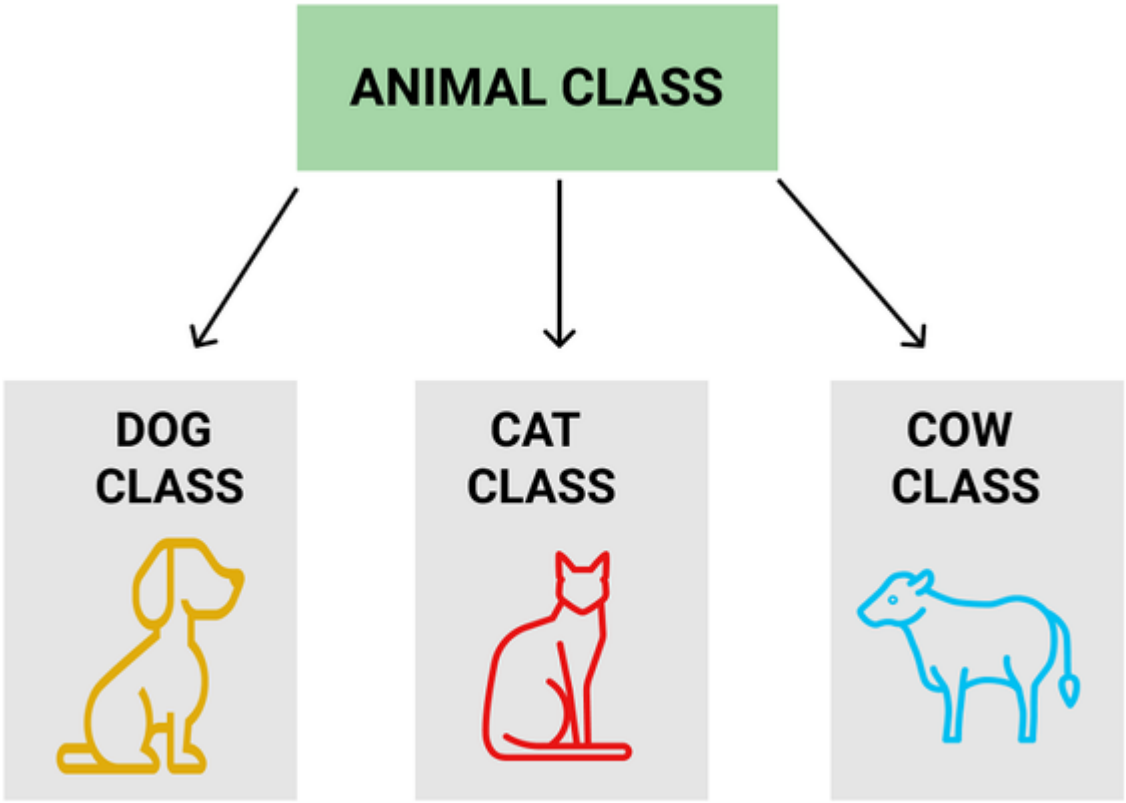
- A compile-time polymorphism feature called **overloading** allows an entity to have numerous implementations of the same name. Method overloading and operator overloading are two examples.
- **Overriding** is a form of run-time polymorphism where an entity with the same name, but a different implementation is executed. It is implemented with the help of virtual functions.

1.5. Polymorphism and Inheritance

- The capability of a class to derive properties and characteristics from another class is called **Inheritance.**
- *Inheritance* is one of the most important features of OOP.
  - **Sub Class:** The class that inherits properties from another class is called: ***Sub Class*** or ***Derived Class.***
  - **Super Class:** The class whose properties are inherited by a sub-class is called: ***Base Class*** or ***Super Class.***
  - **Reusability:** Inheritance supports the concept of "reusability", i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the *fields* and *methods* of the existing class.

1.5. Polymorphism and Inheritance

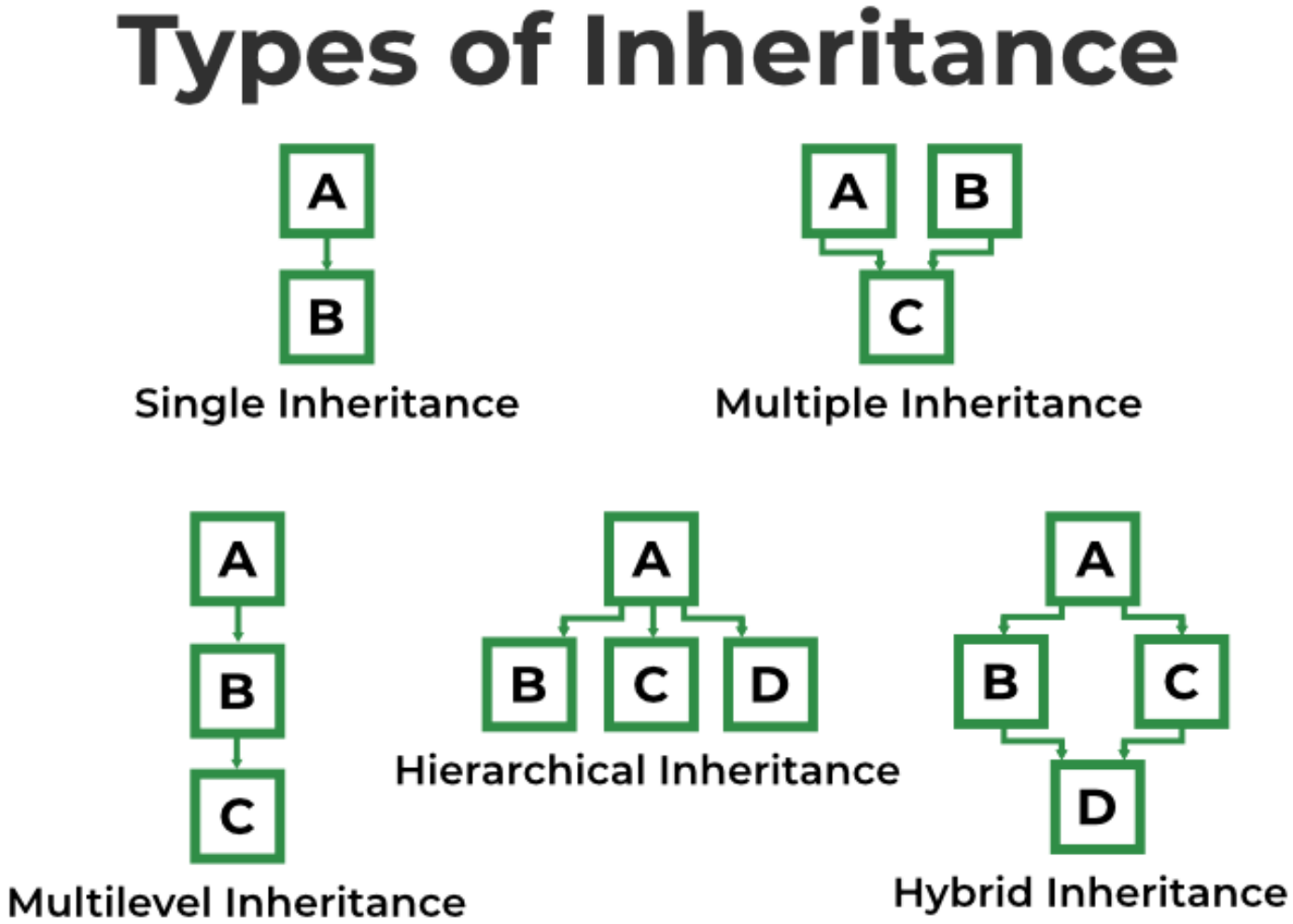
- **Example**: Dog, Cat, Cow can be *Derived Class* of Animal *Base Class*.



1.5. Polymorphism and Inheritance

- *Inheritance* can be classified into 5 types, which are:
  - **Single Inheritance:** Child class derived directly from the base class
  - **Multiple Inheritance:** Child class derived from multiple base classes.
  - **Multilevel Inheritance:** Child class derived from the class which is also derived from another base class.
  - **Hierarchical Inheritance:** Multiple child classes derived from a single base class.
  - **Hybrid Inheritance:** Inheritance consisting of multiple inheritance types of the above specified.

1.5. Polymorphism and Inheritance



1.6. Dynamic Binding

- In **Dynamic binding,** the code to be executed in response to the function call is decided at ***run time***.
- C++ has ***virtual functions*** to support this. Because dynamic binding is flexible, it avoids the drawbacks of **static binding,** which connected the function call and definition at ***build time***.
- Objects communicate with one another by *sending* and *receiving* information. A ***message*** for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. **Message passing** involves specifying the object name, the function name, and the information to be sent.

1.6. Dynamic Binding

- **Example:**

```cpp
// C++ Program to Demonstrate the Concept of Dynamic binding
// with the help of virtual function
#include <iostream>
using namespace std;

class GFG {
public:
    void call_Function() // function that call print
    {
        print();
    }
    void print() // the display function
    {
        cout << "Printing the Base class Content" << endl;
    }
};

class GFG2 : public GFG // GFG2 inherit a publicly
{
public:
    void print() // GFG2's display
    {
        cout << "Printing the Derived class Content"
             << endl;
    }
};
int main()
{
    GFG geeksforgeeks; // Creating GFG's object
    geeksforgeeks.call_Function(); // Calling call_Function
    GFG2 geeksforgeeks2; // creating GFG2 object
    geeksforgeeks2.call_Function(); // calling call_Function
                                    // for GFG2 object
    return 0;
}
```

Output

```
Printing the Base class Content
Printing the Base class Content
```

- As we can see, the print() function of the *parent class* is called even from the *derived class* object. To resolve this we use virtual functions.