OPENTECHZ

A dream to realize your dream

OOP

Opentechz Pvt Ltd .
By Parthasarathi Swain

Encapsulation in C++

Encapsulation is the process of **binding data and functions** that operate on the data into a single unit — **the class**.

It helps in:

- Protecting data from unauthorized access
- Achieving data hiding
- Making code more modular and secure

Real-Life Example:

Think of a **bank account**:

You can **deposit** or **withdraw** money,

But you **can't directly access** the balance — it's hidden inside.

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

class BankAccount {
  private:
    int balance;

  public:
    void deposit(int amount) {
      if(amount > 0)
        balance += amount;
    }

    int getBalance() {
      return balance;
    }
};
```

```
int main() {
  BankAccount acc;
  acc.deposit(1000);
  cout << "Balance: " << acc.getBalance();
//Output: 1000

  return 0;
}
```

Abstraction in C++

Definition:
Abstraction is the process of **hiding internal implementation details** and showing only the **essential features** of an object.

Why Use Abstraction?

- Reduces complexity
- Increases security
- Focuses on what an object does instead of how it does it

```
#include <iostream>
using namespace std;
class BankAccount {
  private:
    int balance;

  public:
    BankAccount() {
      balance = 1000;
    }
    void deposit(int amount) {
      balance += amount;
    }

    void showBalance() {
      cout << "Current Balance: " <<
balance << endl;
    }
};
```

```
int main() {
  BankAccount acc;
  acc.deposit(500);
  acc.showBalance();
  return 0;
}
```

What is Inheritance?

- One class **inherits** features (methods & variables) of another
- Promotes **code reusability**
- Helps in building **hierarchies**
- Reused class = **Base class / Parent class**
- New class = **Derived class / Child class**

Real-life Example:
A **child inherits** properties from parents – name, behavior, etc.

Why Use Inheritance?

- Avoids code duplication
- Enables **code extension**
- Supports **polymorphism**
- Makes maintenance easier
- Encourages **modular programming**

Syntax of Inheritance

```
class Base {
  // base class members
};

class Derived : access_modifier Base {
  // derived class members
};
```

Access Modifiers:

- public
- protected
- private

Types of Inheritance in C++

- ✓ Single Inheritance
- ✓ Multilevel Inheritance
- ✓ Hierarchical Inheritance
- ✓ Multiple Inheritance
- ✓ Hybrid Inheritance

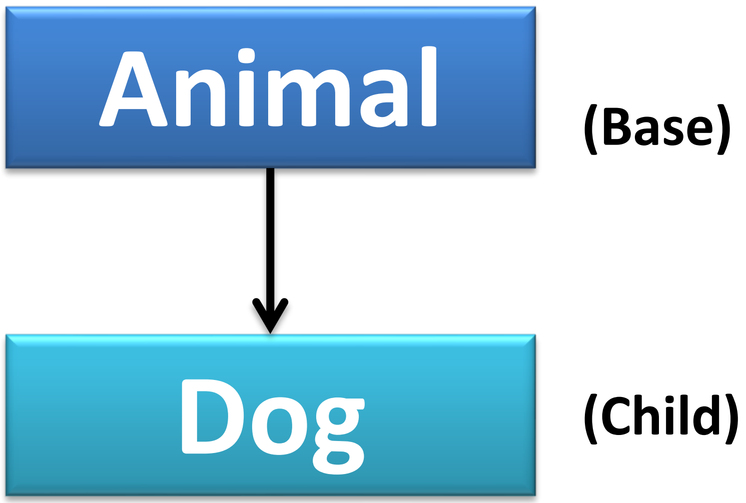
Single Inheritance

- One base class, one derived class
- Most basic form

Example :

```
class Animal {
  void eat();
};

class Dog : public Animal {
  void bark();
};
```



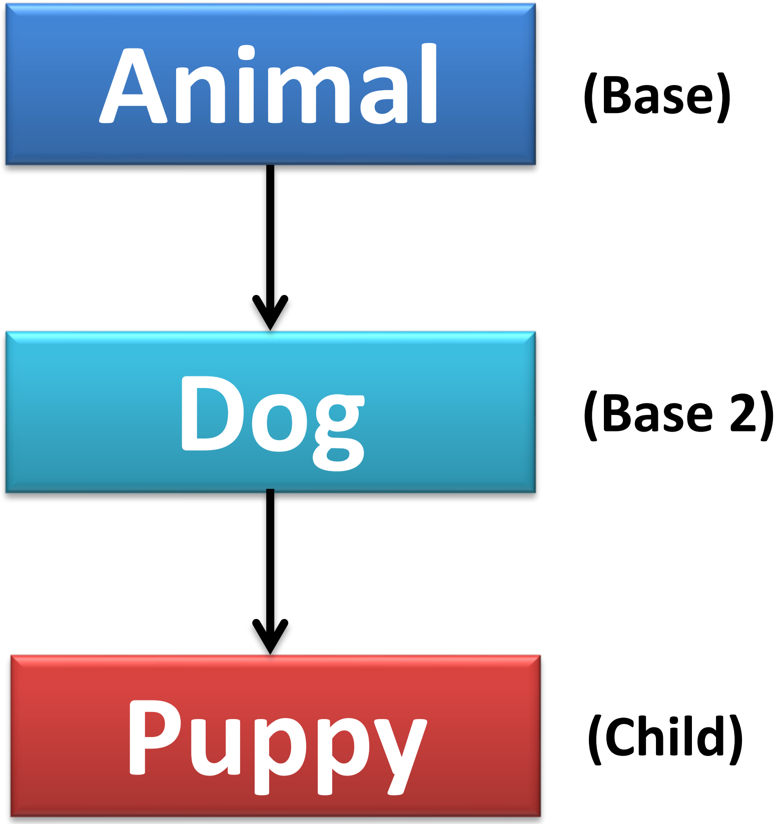
Multilevel Inheritance

- A class is derived from a derived class
- Forms a **chain of inheritance**

Example :

```
class Animal {
  void eat();
};

class Dog : public Animal {
  void bark();
};
class Puppy : public Dog {
  void weep();
};
```



Hierarchical Inheritance

- **Multiple classes** inherit from a **single base class**
- Useful in creating multiple child classes from a common parent

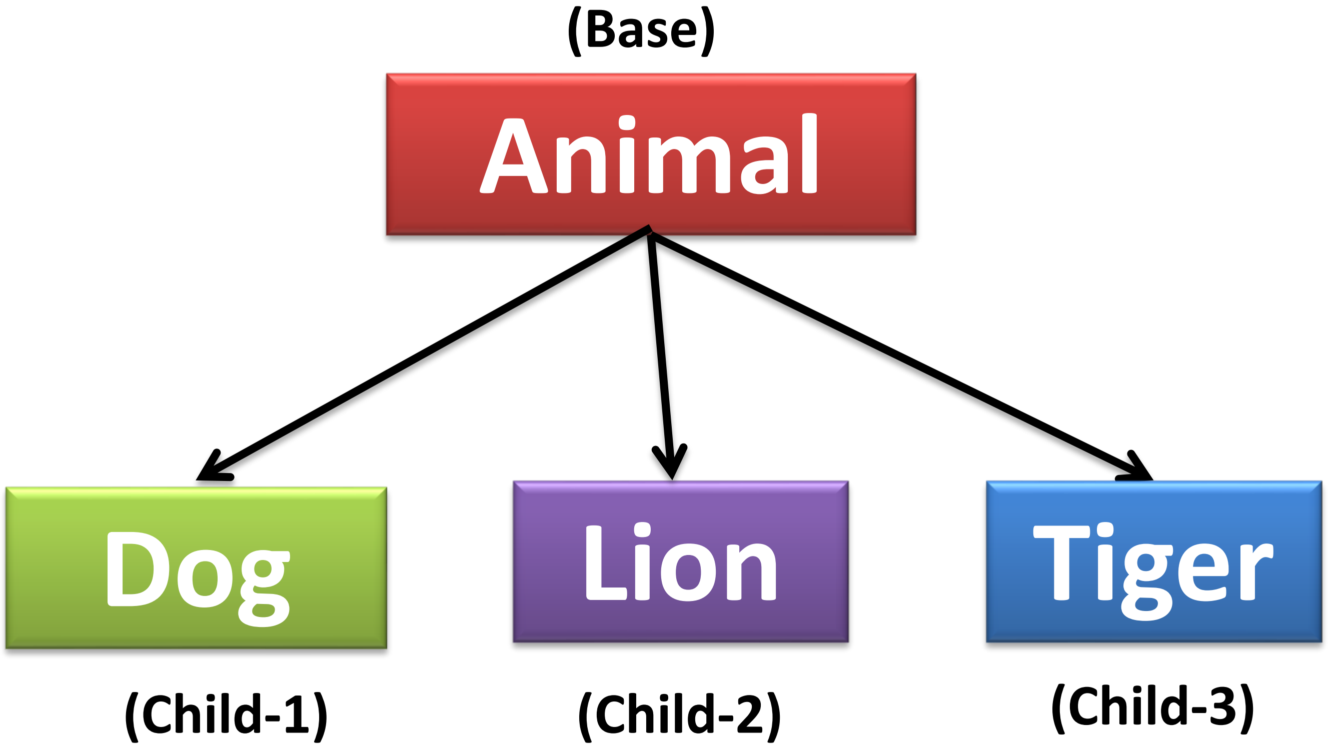
Example :

```
class Animal {
  void eat();
};

class Dog : public Animal {};

class Lion: public Animal {};

class Tiger : public Animal {};
```



Multiple Inheritance

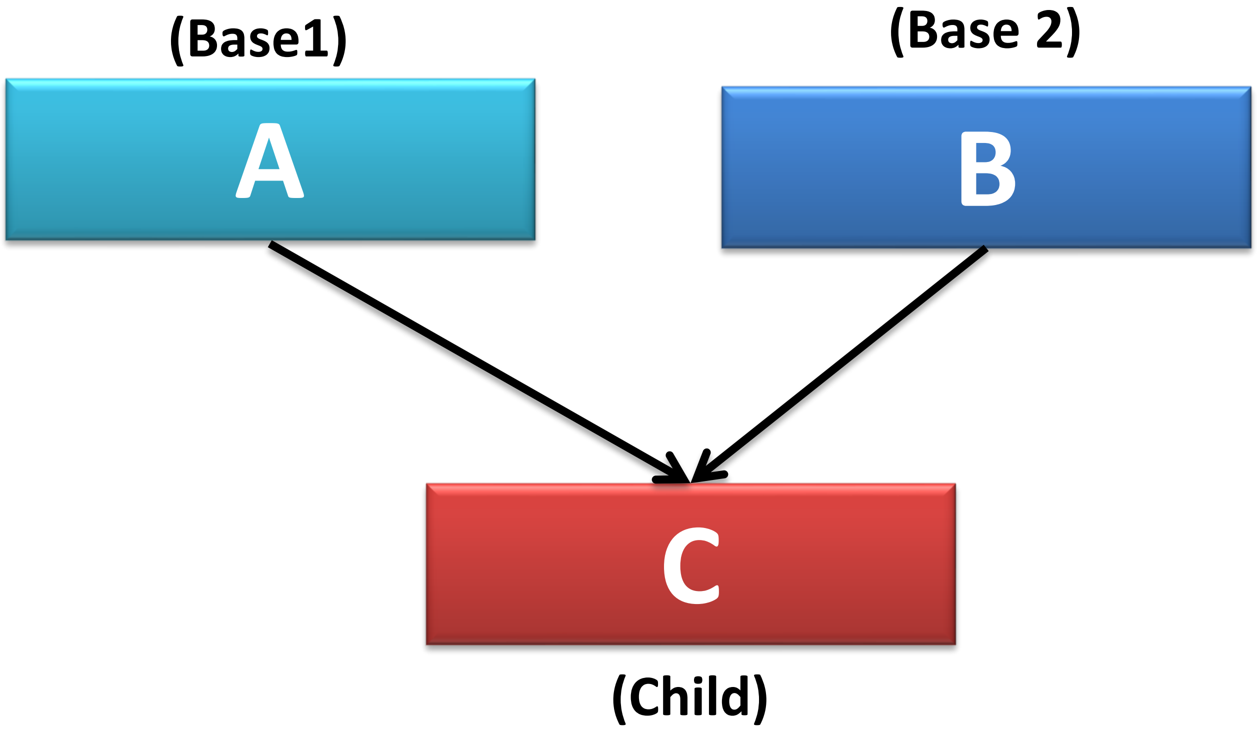
- A class inherits from **more than one base class**
- Can lead to **ambiguity**

Example :

```
class A {
  void show();
};

class B {
  void display();
};

class C : public A, public B {
  // inherits from both
};
```

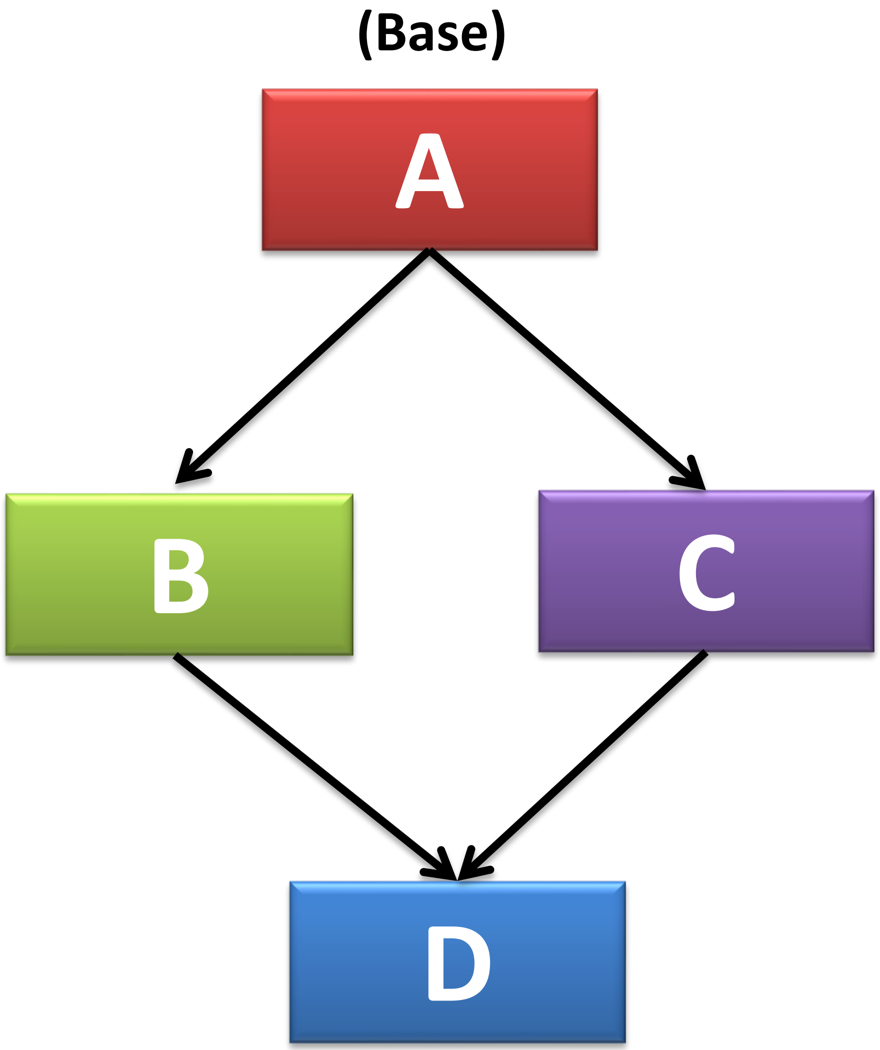


Hybrid Inheritance

- Combines two or more types of inheritance
- There is no particular syntax of hybrid inheritance.

Example :

```
class A {};
class B : public A {};
class C : public A {};
class D : public B, public C {};
```



Polymorphism

- **Polymorphism** means "**many forms**“
- Polymorphism allows **one function, method, or operator** to behave **differently based on the context**.
- It makes **programs more flexible, extensible, and reusable**.

Real-Time Example: SBI ATM

The **SBI ATM** accepts cards from **different banks**, and performs actions accordingly:

- **SBI Card** → Regular Transaction (No Extra Charges)
- **HDFC Card** → Transaction + ₹21 Fee
- **ICICI Card** → Transaction + ₹25 Fee
- **PNB Card** → Limited Withdrawals

➡ **Same SBI ATM**, but behavior **changes** based on the **card’s bank**
➡ This is **Run-Time Polymorphism**

Types of Polymorphism

Compile-time Polymorphism

Achieved using:

✔ Function Overloading

Run-time Polymorphism

Achieved using:

✔ Function Overriding

Function Overloading (Compile-time Polymorphism)

Function Overloading means using the **same function name** with **different parameters** (type or number).

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

class Print {
public:
    void show( int x ) {
        cout << "Integer: " << x << endl;
    }

    void show( string s ) {
        cout << "String: " << s << endl;
    }
};
```

```
int main() {
    Print obj;
    obj.show(10);        // Calls int version
    obj.show("Hello");   // Calls string version
    return 0;
}
```

Operator Overriding (Run-time Polymorphism)

Function Overriding means the **child class** defines a **function with the same name** as in the **parent class**.

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

class Animal {
public:
    void speak() {
        cout << "Animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() {
        cout << "Dog barks" << endl;
    }
};
```

```
int main() {
    Dog d;
    d.speak();  // Output: Dog barks
    return 0;
}
```

OTZ

OPENTECHZ

A dream to realize your dream

Thank You