* + Static data Member and Function
  + Encapsulation
  + Full Encapsulation
  + Why Encapsulation?
  + Advantages of Encapsulation
  + How to achieve Encapsulation
  + Code / Implementation of Encapsulation
  + Real world example of Encapsulation
  + ----------

Encapsulation in C++  
**Encapsulation** is one of the core concepts of Object-Oriented Programming (OOP). It refers to the bundling of data (variables) and methods (functions) that manipulate the data into a single unit, typically a class. The key idea is to hide the internal state of an object and restrict access to only those methods that are intended to be accessed. This is achieved using access specifiers like private, protected, and public.  
**Encapsulation**   
Encapsulation refers to bundling data and the methods that operate on that data into a single unit. Many programming languages use encapsulation frequently in the form of classes. A class is an example of encapsulation in computer science in that it consists of data and methods that have been bundled into a single unit.  
   
Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object. Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.  
   
In other words: Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.  
   
**1. Encapsulation**Encapsulation is about hiding the implementation details from the outside world. In C++, this is done by keeping the data members private and exposing the methods that can manipulate this data as public.  
**2. Full Encapsulation**Full encapsulation occurs when **all** data members of a class are private and can only be accessed or modified via public methods. This ensures full control over the data and maintains integrity.  
**3. Why Encapsulation?**Encapsulation enforces a clear separation between an object's internal state and its external behavior. This is important for several reasons:

* + **Data Protection**: It restricts direct access to the data, minimizing the risk of accidental modifications.
  + **Maintainability**: Code becomes more modular, making it easier to manage and update.
  + **Flexibility**: Changes to the internal implementation can be made without affecting the external code that uses the class.
  + **Control**: You can add validation logic to control how data is accessed and modified.
* **4. Advantages of Encapsulation**
  + **Improved Security**: By restricting direct access to an object's data, encapsulation reduces the chance of unintended interference.
  + **Easier Debugging**: If something goes wrong, you only need to debug the class's public methods.
  + **Data Hiding**: Encapsulation allows the internal state of an object to be hidden from the outside.
  + **Better Control**: Provides a controlled interface for interacting with the object's data.
* **5. How to Achieve Encapsulation in C++**Encapsulation is achieved by using:
  + **Private or Protected** access specifiers for data members.
  + **Public** methods for accessing and modifying these data members (getter and setter functions).
* Encapsulation in C++ is implemented using classes, where the data (attributes) and the methods (functions) are bundled together. Access to the data is restricted using access specifiers like private, protected, and public. Here's an example that demonstrates encapsulation:  
     
     
  #include <iostream>  
  using namespace std;  
  // Class implementing encapsulation  
  class BankAccount {  
  private:  
   // Private data members, only accessible within the class  
   int accountNumber;  
   double balance;  
  public:  
   // Constructor to initialize account details  
   BankAccount(int accNum, double bal) {  
   accountNumber = accNum;  
   balance = bal;  
   }  
   // Public method to deposit money, allows controlled modification of data  
   void deposit(double amount) {  
   if (amount > 0) {  
   balance += amount;  
   cout << "Deposit successful! New balance: $" << balance << endl;  
   } else {  
   cout << "Invalid deposit amount!" << endl;  
   }  
   }  
   // Public method to withdraw money, ensures balance isn't misused  
   void withdraw(double amount) {  
   if (amount > 0 && amount <= balance) {  
   balance -= amount;  
   cout << "Withdrawal successful! New balance: $" << balance << endl;  
   } else {  
   cout << "Invalid withdrawal amount or insufficient balance!"<< endl;  
   }  
   }  
   // Getter for balance, provides controlled read-only access  
   double getBalance() const {  
   return balance;  
   }  
   // Getter for account number  
   int getAccountNumber() const {  
   return accountNumber;  
   }  
  };  
  int main() {  
   // Create a BankAccount object  
   BankAccount account(123456, 500.0);  
   // Access data using public methods  
   cout << "Account Number: " << account.getAccountNumber() << endl;  
   cout << "Initial Balance: $" << account.getBalance() << endl;  
   account.deposit(200.0); // Deposit money  
   account.withdraw(150.0); // Withdraw money  
   return 0;  
  }  
  **7. Real-World Example of Encapsulation**A **bank account** is a classic real-world example of encapsulation:
  + The balance of the account is private and cannot be directly accessed.
  + To access the balance, you need to use methods like checkBalance() (getter).
  + To modify the balance, you need to use methods like deposit() and withdraw() (setters).
  + These methods ensure that operations like withdrawal are valid and prevent direct manipulation of the balance.

**Interview Explanation**Encapsulation is a fundamental OOP concept that bundles data and methods operating on that data into a class while restricting access to the internal details. Full encapsulation ensures that all data is private and can only be accessed via public methods. The goal of encapsulation is to protect the data from unauthorized access and modifications, making the code easier to maintain and debug.