Account.h

```
#pragma once
#include <string>
using namespace std;

// Base class representing a generic bank account
class Account
{
private:
    string m_name;              // Account holder name
    int m_AccNo;                // Unique account number
    static int s_AccNoGenerator;    // Static account number generator

protected:
    float m_balance;            // Current account balance

public:
    // Constructor & Destructor
    Account(const string& name, float balance);
    virtual ~Account();

    // Getter functions
    const string& getName() const;
    float getBalance() const;
    int getAccNo() const;

    // Virtual functions for polymorphic behavior
    virtual void AccumulateInterest();
    virtual void Withdraw(float amount);
    virtual float getInterestRate() const;

    // Common function for all account types
    void Deposit(float amount);
};
```

Function	Type	Description
Account(const string&, float)	Constructor	Initializes account holder’s name and balance, and assigns unique account number.
~Account()	Virtual Destructor	Ensures proper cleanup when deleting derived class objects.
getName()	Getter	Returns the name of the account holder.
getBalance()	Getter	Returns the current balance.
getAccNo()	Getter	Returns the account number.
Deposit(float amount)	Regular Function	Adds the specified amount to the balance.
Withdraw(float amount)	Virtual Function	Base version for withdrawal; can be overridden for specific rules (e.g., minimum balance).
AccumulateInterest()	Virtual Function	Placeholder for interest calculation; implemented in derived classes like Savings.
getInterestRate()	Virtual Function	Returns interest rate (overridden in Savings).

Savings.h

```
#pragma once
#include "Account.h"
#include <string>

// Derived class representing a Savings Account
class Savings : public Account
{
private:
    float m_Rate;    // Interest rate for the savings account

public:
    // Constructor & Destructor
    Savings(const std::string& name, float balance, float rate);
    ~Savings();

    // Overridden virtual functions
    float getInterestRate() const override;
    void AccumulateInterest() override;
};
```

Function	Type	Description
Savings(const std::string& name, float balance, float rate)	Constructor	Initializes account name, balance, and interest rate. Calls Account constructor for base setup.
~Savings()	Destructor	Handles cleanup when a Savings object is destroyed.
float getInterestRate() const override	Overridden Function	Returns the current interest rate for the account.
void AccumulateInterest() override	Overridden Function	Adds interest to the balance based on the stored rate.

Member	Type	Access	Description
m_Rate	float	private	Stores the interest rate (percentage) applied to the savings balance.

Concept	Description
Inheritance	Savings inherits from Account, reusing its data and functions while adding new features.
Polymorphism (Runtime)	Uses override keyword to redefine base class functions for Savings-specific logic.
Encapsulation	Keeps the interest rate private (m_Rate) and provides a controlled way to access it via getInterestRate().
Code Reusability	Leverages the structure of Account but customizes interest accumulation behavior.

Transaction.h

```
#pragma once
#include "Account.h"

// Function to perform a series of transactions on an Account object.
// Demonstrates polymorphism by accepting a base-class pointer.
void Transct(Account*  pAccount);
```

Checking.h

```
#pragma once
#include "Account.h"

// Derived class representing a Checking Account
class Checking : public Account
{
public:
   // Constructors & Destructor
   Checking(const string& name, float balance);
   Checking();
   ~Checking();

   // Overridden function to apply minimum balance rule
   void Withdraw(float amount) override;
};
```

Function	Type	Description
Checking(const string& name, float balance)	Constructor	Initializes a new checking account with account holder name and balance.
Checking()	Default Constructor	Allows creation of an empty object (useful for arrays or dynamic allocation).
~Checking()	Destructor	Handles cleanup when a Checking object is destroyed.
void Withdraw(float amount) override	Overridden Function	Enforces minimum balance rule while withdrawing funds.

□ Inherits Deposit() and general account logic from Account

□ Redefines Withdraw() to add checking-specific constraints

```
}
```

Account.cpp

```
#include "Account.h"
#include "Savings.h"
#include <iostream>
using namespace std;

// Initialize static account number generator
int Account::s_AccNoGenerator  = 1000;

// Constructor
Account::Account(const  string& name, float balance)
    : m_name(name), m_balance(balance)
{
    m_AccNo = ++s_AccNoGenerator;
}

// Destructor
Account::~Account()
{
}

// Getter for account holder name
const string& Account::getName() const
{
    return m_name;
}

// Getter for account balance
float Account::getBalance() const
{
    return m_balance;
}

// Getter for account number
int Account::getAccNo() const
```

```
{
    return m_AccNo;
}

// Base implementation: no interest by default
void Account::AccumulateInterest()
{
}

// Withdrawal function with balance validation
void Account::Withdraw(float amount)
{
    if (amount > m_balance)
    {
        cout << "Insufficient Balance";
    }
    else
    {
        m_balance -= amount;
    }
}

// Deposit function
void Account::Deposit(float amount)
{
    m_balance += amount;
}

// Default interest rate (0 for generic account)
float Account::getInterestRate() const
{
    return 0.0f;
}
```

Functionality	Description
Manage account data	Stores account holder’s name, balance, and unique account number.
Provide core operations	Supports deposit, withdrawal, and balance inquiry.
Serve as a base class	Allows derived classes (Savings, Checking) to override specific behaviors.

Static Data Member

```
int Account::s_AccNoGenerator  = 1000;
```

- Used to **generate unique account numbers** for each account.
- Each new account increments this value to ensure uniqueness.

◆ Constructor

```
Account::Account(const string& name, float balance)
   : m_name(name), m_balance(balance)
{
   m_AccNo = ++s_AccNoGenerator;
}
```

- Initializes account holder’s name and balance.
- Automatically assigns a **unique account number**.

◆ Destructor

```
Account::~Account() {}
```

- Simple destructor (no dynamic allocation here).
- Declared virtual to ensure proper cleanup in derived classes.

◆ Getter Functions

```
const string& Account::getName() const { return m_name; }
float Account::getBalance() const { return m_balance; }
int Account::getAccNo() const { return m_AccNo; }
```

- Provide **read-only access** to private data members.
- Promote **encapsulation** by preventing direct modification.

◆ Deposit Function

```
void Account::Deposit(float amount)
{
   m_balance += amount;
}
```

- Adds the deposited amount to the account balance.
- Common to all account types (not overridden).

◆ Withdraw Function

```
void Account::Withdraw(float amount)
{
   if (amount > m_balance)
      cout << "Insufficient Balance";
   else
      m_balance -= amount;
}
```

- Ensures **no overdraft** occurs.
- Can be **overridden** (e.g., Checking adds minimum balance rules).

◆ Interest Accumulation

```
void Account::AccumulateInterest() {}
```

- Empty in base class since not all accounts earn interest.
- Savings overrides this to calculate interest.

◆ Interest Rate Getter

```
float Account::getInterestRate() const
{
   return 0.0f;
}
```

- Default: 0% interest for base class.
- Overridden by Savings to return the actual rate.

🧩 Inheritance Structure

```
Account (Base)
 ├── Savings  → Adds interest rate & accumulation
 └── Checking → Adds withdrawal rules
```

Checking.cpp

```
#include "Checking.h"
#include <iostream>
using namespace std;

// Constructor
Checking::Checking(const string& name, float balance)
    : Account(name, balance)
{
}

// Destructor
Checking::~Checking()
{
}

// Overridden Withdraw method for Checking account
void Checking::Withdraw(float amount)
{
    if ((m_balance - amount) > 50.0f)
    {
        Account::Withdraw(amount);
    }
    else
    {
        cout << "Cannot withdraw amount. Minimum balance should remain 50 after withdrawal." << endl;
    }
}
```

🧩 Purpose

Checking is a **derived class** from Account that models a **Checking Account**.
It overrides the base class's Withdraw() function to enforce a **minimum balance rule**, ensuring that the account balance never drops below ₹50 after withdrawal.

⚙️ Implementation Overview

🔹 Constructor

```
Checking::Checking(const string& name, float balance)
   : Account(name, balance)
{
}
```

- Initializes a Checking account using the **base class constructor**.
- Inherits account number generation, balance, and holder name setup from Account.

🔹 Destructor

```
Checking::~Checking() {}
```

- A simple destructor since there's no dynamic allocation.
- Automatically calls the base class destructor when destroyed.

🔹 Withdraw Function (Overridden)

```
void Checking::Withdraw(float amount)
{
   if ((m_balance - amount) > 50.0f)
   {
      Account::Withdraw(amount);
   }
   else
   {
      cout << "Cannot withdraw amount. Minimum balance should remain 50 after withdrawal." << endl;
   }
}
```

🧠 Explanation:

- Ensures that after withdrawing, at least **₹50 remains in the account**.
- If the rule is satisfied → calls the **base class withdrawal** logic.
- If not → displays a message and **prevents** the transaction.

Saving.cpp

```
#include "Savings.h"


// Constructor
Savings::Savings(const std::string& name, float balance, float rate)
    : Account(name, balance), m_Rate(rate)
{
}


// Destructor
Savings::~Savings() {}


// Overridden getter for interest rate
float Savings::getInterestRate() const
{
    return m_Rate;
}


// Overridden function to accumulate interest
void Savings::AccumulateInterest()
{
    m_balance += (m_balance * m_Rate) / 100.0f;
}
```

🧩 Purpose

Savings is a **derived class** from the base class Account.
It represents a **Savings Account** that **earns interest** based on a given rate.
This class overrides certain virtual functions from Account to add interest-specific logic.

⚙️ Implementation Overview

🔹 Constructor

```
Savings::Savings(const std::string& name, float balance, float rate)
    : Account(name, balance), m_Rate(rate)
{
}
```

🧠 **Explanation:**

- Calls the **base class constructor** to initialize the account holder’s name and starting balance.
- Also initializes the **interest rate (m_Rate)** specific to savings accounts.
- Uses **member initializer list** syntax for efficient initialization.

✅ **Example:**

```
Savings sav("Shivam", 1000.0f, 5.0f);
```

Creates a savings account with:

- Name → “Shivam”
- Initial Balance → ₹1000
- Interest Rate → 5%

🔹 Destructor

```
Savings::~Savings() {}
```

- Simple destructor since there’s no dynamic memory allocation.
- Automatically calls Account destructor when the object goes out of scope.

🔹 Overridden: getInterestRate()

```
float Savings::getInterestRate() const
{
   return m_Rate;
}
```

🧠 **Explanation:**

- Returns the **interest rate** applicable to this savings account.
- Overrides the base class version (which returns 0.0f by default).
- Allows **polymorphic access** — meaning even if accessed via a base pointer (Account*), it returns the correct savings rate.

🔹 Overridden: AccumulateInterest()

```
void Savings::AccumulateInterest()
{
   m_balance += (m_balance * m_Rate) / 100.0f;
}
```

🧠 **Explanation:**

- Adds **interest to the account balance**.
- Formula:

$$\text{New Balance} = \text{Old Balance} + \frac{\text{Old Balance} \times \text{Rate}}{100}$$

- Example:
  - If balance = ₹1000 and rate = 5%,
    then new balance = ₹1000 + (₹1000 × 5 / 100) = ₹1050.

🧩 **Concept:**

This demonstrates **polymorphism** — when a base class pointer calls AccumulateInterest(), the Savings version executes automatically.

🧩 **Inheritance Relationship**

Account (Base Class)

```
  ↑
  └── Savings (Derived Class)
```

- Inherits all common banking features (balance, deposit, withdraw).
- Adds its own interest mechanism

Transaction.cpp

```
#include "Transaction.h"
#include <iostream>
using namespace std;

// Function to perform a series of account transactions
void Transct(Account* pAccount)
{
    cout << "Transaction Started" << endl;
    cout << "Initial Balance: " << pAccount->getBalance() << endl;

    // Deposit step
    float depositAmount = 200.0f;
    pAccount->Deposit(depositAmount);
    cout << "Deposited: " << depositAmount
        << " -> Balance Now: " << pAccount->getBalance() << endl;

    // Withdraw step
    float withdrawAmount = 170.0f;
    pAccount->Withdraw(withdrawAmount);
    cout << "Withdrawn: " << withdrawAmount
         << " -> Balance Now: " << pAccount->getBalance() << endl;

    // Interest accumulation step
    pAccount->AccumulateInterest();
    cout << "Interest Added (" << pAccount->getInterestRate()
         << "%) -> Balance Now: " << pAccount->getBalance() << endl;

    // Final summary
    cout << "Final Balance: " << pAccount->getBalance() << endl;
    cout << "--------------------------------------------" << endl;
}
```

🧩 Purpose

This file defines the **Transct()** function, which performs a sequence of banking operations on any account type (e.g., Savings, Checking).
It uses **runtime polymorphism** by accepting a pointer to the **base class Account**, allowing dynamic behavior depending on the actual object type passed.

⚙️ Implementation Overview

🔹 Function Definition

```
void Transct(Account* pAccount)
```

🧠 Explanation:

- Accepts a **pointer to an Account object** — can point to any derived type (Savings, Checking, etc.).
- Enables **polymorphism**: calls to virtual functions like Withdraw() or AccumulateInterest() will invoke the correct version depending on the object type.

🔹 Step 1 — Display Initial Balance

```
cout << "Initial Balance: " << pAccount->getBalance() << endl;
```

Shows the account’s current balance before any operations begin.

🔹 Step 2 — Deposit

```
float depositAmount = 200.0f;
pAccount->Deposit(depositAmount);
cout << "Deposited: " << depositAmount
     << " -> Balance Now: " << pAccount->getBalance() << endl;
```

🧠 **Explanation:**

- Adds ₹200 to the balance using the Deposit() method.
- Deposit() is a **non-virtual base function**, common to all account types.

🔹 Step 3 — Withdraw

```
float withdrawAmount = 170.0f;

pAccount->Withdraw(withdrawAmount);

cout << "Withdrawn: " << withdrawAmount
    << " -> Balance Now: " << pAccount->getBalance() << endl;
```

🧠 **Explanation:**

- Withdraws ₹170 from the account.
- If pAccount points to:
  - A **Checking account**, the overridden version applies **minimum balance rules**.
  - A **Savings account**, it allows normal withdrawal.

This demonstrates **runtime polymorphism** in action.

🔹 Step 4 — Apply Interest

```
pAccount->AccumulateInterest();

cout << "Interest Added (" << pAccount->getInterestRate()
    << "%) -> Balance Now: " << pAccount->getBalance() << endl;
```

🧠 **Explanation:**

- Calls AccumulateInterest(), which:
  - Does nothing for base Account.
  - Adds interest for Savings accounts (based on their rate).
- Uses getInterestRate() (virtual) to display the correct rate dynamically.

🔹 Step 5 — Final Summary

```
cout << "Final Balance: " << pAccount->getBalance() << endl;

cout << "--------------------------------------------"  << endl;
```

Displays the final account balance after all transactions.

Concept	**Description**
Polymorphism	Same function (Transct) works with different account types.
Encapsulation	All operations go through class methods (no direct balance modification).

Concept	Description
Reusability	One transaction function handles multiple account types.
Dynamic Binding	Virtual functions ensure the right derived behavior at runtime.

🏦 Banking System Project – Summary

📘 Overview

This project is a **C++ Object-Oriented Programming (OOP)** implementation of a simple **Banking System**.
It demonstrates **core OOP concepts** such as **inheritance, polymorphism, encapsulation, and abstraction** using real-world banking operations like deposits, withdrawals, and interest calculations.

⚙️ Key Components

1. **Account (Base Class)**
   - Represents a **generic bank account**.
   - Stores common details: account holder name, account number, and balance.
   - Provides core functionalities like deposit, withdrawal, and balance retrieval.
   - Acts as a **base class** for other account types.
2. **Savings (Derived Class)**
   - Inherits from Account.
   - Adds an **interest rate** feature.
   - Overrides AccumulateInterest() and getInterestRate() to calculate and apply interest to the balance.
3. **Checking (Derived Class)**
   - Also inherits from Account.
   - Overrides Withdraw() to include a **minimum balance rule** (₹50 must remain after withdrawal).
4. **Transaction (Utility Function)**
   - A standalone function (Transct(Account* pAccount)) that performs a **series of transactions** — deposit, withdrawal, and interest calculation.
   - Demonstrates **runtime polymorphism**, as it can work with any Account type (Savings or Checking).

🧠 OOP Concepts Demonstrated

Concept	Description
Encapsulation	Account details are kept private; accessed through public methods.
Inheritance	Savings and Checking classes inherit from Account.
Polymorphism	Functions like Withdraw() and AccumulateInterest() behave differently based on the object type.

Concept	Description
Abstraction	Complex account behavior is simplified using class methods.

💻 Program Flow

1. Create an object of Savings or Checking.
2. Call the Transct() function and pass the object.
3. The program performs:
   - Deposit → Withdraw → Interest Accumulation.
4. Outputs the **initial**, **intermediate**, and **final balance** with transaction details.

📊 Sample Output

```
Transaction Started
Initial Balance: 100
Deposited: 200 -> Balance Now: 300
Withdrawn: 170 -> Balance Now: 130
Interest Added (0.5%) -> Balance Now: 130.65
Final Balance: 130.65
--------------------------------------------
```

🧩 Learning Outcome

Through this project, you will:

- Understand **class hierarchy** and **virtual functions**.
- See **real-world use** of OOP principles in financial applications.
- Gain a foundation for **building low-latency or high-performance financial systems** (useful for quant/C++ developer roles).