|  |  |  |  |
| --- | --- | --- | --- |
| **Object Oriented Programming Lab 08** | | | |
| **Course Code:** | COMP-112L | **Class** | CS (B) |
| **Lab Engineer** | Laiba Khalid | **Semester** | 2nd |
| **Lab Title** | C++ Classes | **Section** | - |
| **Name** |  | **Reg no.** |  |
| **Content Covered** | C++ Class Implementation. | | |
| **Instructions:**  • Submit the file with your names following your registration numbers like AI001\_Name.  • Submit soft copy of the report before deadline. Marks will be deducted for late submissions. | | | |

**Data Encapsulation in C++**

**Introduction:**

**Encapsulation** is the technique of bundling the data (variables) and methods (functions) that operate on the data into a single unit, usually a class. It also helps in data hiding by restricting direct access to some data fields.

**Encapsulation in C++:**

Encapsulation is implemented using access specifiers:

* **Private:** Data members are accessible only within the class.
* **Protected:** Data members are accessible within the class and its derived classes.
* **Public:** Data members are accessible from outside the class.

**Getter and Setter Functions:**

Getter and setter functions provide controlled access to private attributes:

* Getter functions return the value of private attributes.
* Setter functions allow modifying private attributes while maintaining validation rules.

**Example**: **Bank Management System**

1. Create a class BankAccount with private attributes balance and accountHolder.
2. Implement getter and setter methods for accountHolder.
3. Implement deposit(), withdraw(), and getBalance() to access and modify the balance securely.

|  |
| --- |
| #include <iostream>  using namespace std;  class BankAccount {  private:  double balance;  string accountHolder;  public:  BankAccount(string holder, double initial\_balance) {  setAccountHolder(holder);  balance = initial\_balance;  }  void deposit(double amount) {  if (amount > 0) {  balance += amount;  cout << "Deposited: " << amount << endl;  } else {  cout << "Invalid deposit amount" << endl;  }  }  void withdraw(double amount) {  if (amount > 0 && amount <= balance) {  balance -= amount;  cout << "Withdrawn: " << amount << endl;  } else {  cout << "Insufficient balance or invalid amount" << endl;  }  }  double getBalance() {  return balance;  }  string getAccountHolder() {  return accountHolder;  }  void setAccountHolder(string holder) {  if (!holder.empty()) {  accountHolder = holder;  } else {  cout << "Invalid account holder name" << endl;  }  }  };  int main() {  BankAccount account("John Doe", 5000);  account.deposit(1500);  account.withdraw(2000);  cout << "Account Holder: " << account.getAccountHolder() << endl;  cout << "Balance: " << account.getBalance() << endl;    account.setAccountHolder("Jane Doe");  cout << "Updated Account Holder: " << account.getAccountHolder() << endl;    return 0;  } |

**Lab Activity:**

1. Modify the BankAccount class to restrict negative balance withdrawals.
2. Create a Student class where name and grades are encapsulated, and provide getter and setter methods to update grades securely.
3. Implement an Employee class that hides salary details but provides controlled access through getter and setter methods.