Object Oriented Programming (Spring2024)

Week03 (26/27/28-Feb-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Access Specifiers

In object-oriented programming (OOP), access specifiers are keywords used to control the accessibility of members (data and functions) within a class. They define the scope from where these members can be accessed and modified, promoting encapsulation and data protection. This lecture focuses further on understanding the concept of access specifiers and their significance in C++.

1. What are Access Specifiers?

Access specifiers are keywords that dictate the visibility and accessibility of class members. They determine which parts of the program can access and modify the data and functions defined within a class. This controlled access plays a crucial role in achieving:

- **Data Encapsulation:** Encapsulation is the principle of bundling data (member variables) and the operations that manipulate that data (member functions) within a single unit, the class. Access specifiers help enforce encapsulation by restricting direct access to internal data, promoting data integrity and consistency.
- **Code Reusability:** By controlling access, we can create reusable classes with well-defined interfaces (public members) and protected implementations (private members). This allows other parts of the program to interact with the class without relying on its internal details, facilitating code reuse and maintainability.

2. Importance of Access Specifiers:

Access specifiers offer several benefits in object-oriented design:

- **Protects Data Integrity:** By restricting direct access to data members, we prevent accidental or unintended modifications from outside the class. This ensures that data remains consistent and adheres to the intended behavior of the class.
- **Enhances Code Maintainability:** By separating the interface (public members) from the implementation (private members), access specifiers promote modularity and improve code readability. Changes to internal implementation details can be made without affecting the public interface, simplifying maintenance and reducing the risk of unintended side effects.
- **Promotes Reusability:** Well-defined access specifiers enable the creation of reusable classes with clear interfaces. Other parts of the program can interact with the class through its public members without needing to understand the internal workings, facilitating code reuse across different parts of the application.

3. Code Example:

Consider a simple `Student` class:

```
class Student {
private:
  int id;
  std::string name;

public:
  // Constructor to initialize data
  Student(int id, const std::string name) : id(id), name(name) {}

  // Public getter method to access name safely
  std::string getName() const { return name; }
};
```

In this example:

- `id` and `name` are declared as `private`, making them inaccessible directly from outside the class. This enforces data encapsulation and protects student information.
- The public constructor allows initializing these private members during object creation.
- The public `getName` method provides a controlled way to access the student's name without compromising data integrity.

4. Conclusion:

Access specifiers are fundamental building blocks in object-oriented design with C++. They enable data encapsulation, promote code maintainability, and facilitate code reusability. Understanding and effectively using access specifiers is essential for creating robust, well-structured, and maintainable C++ applications.

Types of Access Specifiers in C++

Let's dives deeper into the three fundamental access specifiers available in C++:

1. Public Access Specifier:

The `public` access specifier grants access to class members from **anywhere** within the program. This means:

- **Member variables and member functions declared as public can be accessed directly using the dot operator (.) on an object of the class.**
- **They are visible and usable from within the class, other member functions, and even outside the class in the main function or any other part of the program.**

Code Example:

```
class Counter {
public:
  int count; // Public member variable

  // Public constructor to initialize count
  Counter(int initialValue) : count(initialValue) {}

  // Public member function to increment count
  void increment() { count++; }

  // Public member function to access count
  int getCount() const { return count; }
};


int main() {
  Counter c(5); // Create a Counter object
  c.increment(); // is the same as "c.count++;"
  std::cout << c.getCount() << std::endl; // "c.getCount()" is the same a
s "c.count"
  return 0;
}
```

2. Private Access Specifier:

The `private` access specifier restricts access to class members **only within the class definition**. This means:

- **Member variables and member functions declared as private cannot be accessed directly from outside the class.**
- **They are only visible and usable from within the class and its member functions.**

Code Example:

```
class BankAccount {
private:
  int balance; // Private member variable

public:
  // Public constructor to initialize balance
  BankAccount(int initialBalance) : balance(initialBalance) {}

  // Public member function to deposit funds (modifies private balance)
  void deposit(int amount) { balance += amount; }

  // Public member function to get balance (reads private balance)
  int getBalance() const { return balance; }
};



int main() {
  BankAccount account(100); // Create a BankAccount object
  // account.balance = 200; // This is not allowed, balance is private
  account.deposit(50); // Call public function to modify balance indirect
ly
  std::cout << account.getBalance() << std::endl; // Call public function
to access balance
  return 0;
}
```

3. Protected Access Specifier:

The `protected` access specifier grants access to class members from:

- **Within the class itself and its member functions.**
- **Derived classes (classes that inherit from the base class).**

Code Example:

```
class Animal {
protected:
  std::string name;

public:
  Animal(const std::string animalName) : name(animalName) {}

  virtual void speak() const = 0; // Pure virtual function (explained in
inheritance)
};
```

```
class Dog : public Animal {
public:
  Dog(const std::string dogName) : Animal(dogName) {}

  void speak() const override { std::cout << "Woof!" << std::endl; }
};

int main() {
  Dog d("Buddy");
  d.speak(); // Call public function from main
  // d.name = "Rex"; // This is not allowed, name is protected
  return 0;
}
```

Key Points:

- **Default Access Specifier:** If no access specifier is explicitly mentioned, members default to private access in C++.
- **Choosing the Right Specifier:** The appropriate access specifier depends on the intended use of the member:
  - Public members form the class's interface, accessible from anywhere.
  - Private members encapsulate data and are only accessible within the class, promoting data protection.
  - Protected members allow controlled access for inheritance, facilitating code reusability and specialization.

Remember: Effective use of access specifiers is crucial for building robust, maintainable, and secure object-oriented applications in C++.

Declaring and Using Members with Different Access Specifiers

Building upon the understanding of access specifiers and their types, let's delve into the practical aspects of declaring and using members with different access specifiers in C++.

1. Declaring Members with Access Specifiers:

The access specifier keyword is placed **before** the member declaration within the class definition. Here's the syntax:

```
class MyClass {
  // Access specifier member_declaration;
  // ... other members
};
```

Example:

```
class Person {
private:
  std::string name;
  int age;

public:
  Person(const std::string n, int a) : name(n), age(a) {}

  std::string getName() const { return name; }
  int getAge() const { return age; }
};
```

In this example:

- `name` and `age` are declared as `private`, restricting direct access from outside the class.
- `getName` and `getAge` are declared as `public`, allowing access through the object's dot operator.

2. Using Public Members:

Public members act as the interface of the class, accessible from anywhere in the program using the dot operator (.) on an object of the class.

```
Person p("Alice", 30);
std::cout << p.getName() << " is " << p.getAge() << " years old." << st
d::endl;
```

3. Using Private Members (Indirectly):

Private members are not directly accessible from outside the class. However, we can provide public member functions (accessors or mutators) to interact with them indirectly:

```
class Account {
private:
  double balance;

public:
  Account(double initialBalance) : balance(initialBalance) {}

  void deposit(double amount) { balance += amount; }
  double getBalance() const { return balance; }
};
```

```
int main() {
  Account acct(1000.0);
  acct.deposit(500.0);
  std::cout << "Balance: $" << acct.getBalance() << std::endl;
  return 0;
}
```

4. Protected Members (Covered in Inheritance):

Protected members, discussed in the context of inheritance, are accessible within the class, its member functions, and derived classes. We will explore their usage in detail when covering inheritance.

Key Points:

- Choose the appropriate access specifier based on the member's intended use and visibility requirements.
- Public members form the class's public interface, accessible from anywhere.
- Private members are encapsulated and only accessible indirectly through public member functions.
- Protected members (explained in inheritance) allow controlled access for derived classes.
- Always strive for clarity and maintainability by explicitly specifying access specifiers for all class members.

By effectively declaring and using members with different access specifiers, you can achieve robust data encapsulation, promote code maintainability, and build secure and well-structured object-oriented applications in C++.

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Choosing the Appropriate Access Specifier in C++

Selecting the right access specifier for each member in your C++ class is crucial for achieving data encapsulation, promoting code maintainability, and facilitating reusability. Here's a comprehensive guide to making informed decisions:

1. Understanding the Member's Role:

The primary factor influencing your choice is the member's intended purpose and usage within the class and its potential interactions with other parts of the program. Consider these questions:

- **Is the member part of the class's public interface, accessible from anywhere?**
- **Does the member need to be hidden and protected from direct modification?**
- **Is the member intended for use by derived classes (inheritance)?**

2. Guidelines for Choosing Access Specifiers:

Based on the member's role, here are general recommendations:

- **Public:**
  - Members that form the class's public interface, accessible from anywhere in the program using the dot operator (.).
  - Examples: Public constructors, member functions providing access to data or performing core functionalities.
- **Private:**
  - Members that encapsulate the class's internal implementation details and data, protecting them from direct modification outside the class.
  - Examples: Data members representing the class's state, helper functions used internally by other member functions.
- **Protected:**
  - Members intended for controlled access within the class, its member functions, and derived classes (covered in inheritance).
  - Examples: Base class members that derived classes need to access or modify in specialized ways.

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Optional Material

Reflection MCQs:

1. **What is the main purpose of access specifiers in C++?**

   - A. To define data members
   - B. To control the visibility and access of class members
   - C. To enable polymorphism
   - D. To set default values for members

   Click to reveal the answer
2. **Which access specifier allows a class member to be accessible from outside the class?**

   - A. public
   - B. private
   - C. protected
   - D. friend

   Click to reveal the answer

3. **In C++, what does the `protected` access specifier indicate?**

   - A. Members are accessible only within the same class
   - B. Members are accessible from any class
   - C. Members are accessible only within the same class and its subclasses
   - D. Members are accessible only within the same file

   Click to reveal the answer
4. **What is the role of access specifiers in achieving data encapsulation?**

   - A. To hide the implementation details of a class
   - B. To provide default values for class members
   - C. To control the lifetime of objects
   - D. To facilitate function overloading

   Click to reveal the answer

5. **Why is it important to control access to class members using access specifiers?**

   - A. To reduce memory consumption

- B. To improve code readability
- C. To enhance security and prevent unauthorized access
- D. To speed up program execution

Click to reveal the answer

6. **What does the `public` access specifier indicate in C++?**

   - A. Members are accessible only within the same class
   - B. Members are accessible from any class
   - C. Members are accessible only within the same file
   - D. Members are accessible only within the same package

   Click to reveal the answer

7. **In C++, which access specifier allows access only within the same class and its subclasses?**

   - A. public
   - B. private
   - C. protected
   - D. friend

   Click to reveal the answer

8. **What is the default access specifier for members of a C++ class?**

   - A. public
   - B. private
   - C. protected
   - D. It depends on the compiler

   Click to reveal the answer

9. **Which of the following statements is true about private access specifier in C++?**

   - A. Members are accessible only within the same class
   - B. Members are accessible from any class
   - C. Members are accessible only within the same file
   - D. Members are accessible only within the same package

   Click to reveal the answer

Access Specifiers and their Relation with Accessor/Mutator Methods

In object-oriented programming with C++, access specifiers and accessor/mutator methods (also known as getter and setter methods) work hand-in-hand to achieve data encapsulation and control member access within a class.

1. Accessor/Mutator Methods:

Accessor and mutator methods are member functions within a class that provide controlled access to private members:

- **Getter methods (Accessors):** These methods allow **reading** the value of a private member variable. They are typically declared as `public` to grant access from outside the class.
- **Setter methods (Mutators):** These methods allow **modifying** the value of a private member variable. They can be declared as `public` or `private` depending on the desired level of control.

2. Relationship between Access Specifiers and Accessor/Mutator Methods:

Access specifiers determine the visibility of both the member variable and the accessor/mutator methods that operate on it:

- **Private Member:**
  - Accessible only within the class definition.
  - Requires both getter and setter methods declared as `public` to provide controlled access

```
class Counter {
private:
  int count;

public:
  // Public getter to access count
  int getCount() const { return count; }

  // Public setter to modify count
  void setCount(int newCount) { count = newCount; }
};
```

- **Protected Member:**
  - Accessible within the class, its member functions, and derived classes (when inheritance is introduced).
  - May or may not require accessor/mutator methods depending on the intended access level within the class hierarchy.

4. Benefits of Using Accessor/Mutator Methods:

- **Enforced Data Validation:** Setter methods can perform validation checks on input values before modifying the private member, ensuring data integrity and consistency.
- **Centralized Control:** Accessor/mutator methods encapsulate the logic for accessing and modifying data within the class, promoting code maintainability and reducing the risk of unintended side effects.
- **Flexibility:** You can choose to make setter methods `private` to restrict direct modification from outside the class, further enhancing data protection.

5. Code Example:

```
class BankAccount {
private:
  double balance;

public:
  // Public constructor with initial balance
  BankAccount(double initialBalance) : balance(initialBalance) {}

  // Public getter to access balance (read-only)
  double getBalance() const { return balance; }

  // Public setter to deposit funds (modifies balance)
  void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
    } else {
      std::cerr << "Error: Deposit amount cannot be negative." << std::en
dl;
    }
  }
};
```

In this example:

- `balance` is private, enforcing data encapsulation.
- `getBalance` allows read-only access to the balance.
- `deposit` validates and modifies the balance while encapsulating the logic within the class.

6. Conclusion:

By effectively combining access specifiers and accessor/mutator methods, you can achieve robust data encapsulation, promote code maintainability, and build secure and well-structured object-oriented applications in C++. Remember to choose the appropriate access specifier and accessor/mutator approach based on the intended use and access requirements for each member within your class.

In-Class Project: Student Result Calculation

Project Description:

Welcome to the Student Results System project! This hands-on exercise will deepen your understanding of fundamental Object-Oriented Programming (OOP) concepts. Building on the basics of classes, objects, data members, methods, getters, mutators, constructors, and access specifiers, this project introduces additional functionality to create a more robust student information management system.

Project Overview:

In this project, you will create a `Student` class that represents a student and manages both basic information (name, roll number) and additional features like subject-related information (number of subjects, subject names, subject grades). The system will allow you to set and retrieve this information and calculate an overall grade based on subject grades.

Sequence of Steps:

1. **Define the `Student` Class (student.h):**
    - Start by defining the structure of the `Student` class in the header file (`student.h`).
    - Include data members for basic student information and subject-related information.
    - Implement the necessary methods, getters, setters, and the method to calculate the overall grade.
        - **Data Members:**
            - `name` (string): Represents the student's name.
            - `rollNumber` (int): Represents the student's roll number.
            - `numSubjects` (int): Represents the number of subjects the student is enrolled in.
            - `subjectNames` (vector of strings): Stores the names of the subjects.
            - `subjectGrades` (vector of chars): Stores the grades for each subject.
        - **Constructors:**
            - `Student()` : Default constructor initializes default values for `name` , `rollNumber` , `numSubjects` , `subjectNames` , and `subjectGrades` .
            - `Student(const std::string& studentName, int roll)` : Overloaded constructor with basic student information.
            - `Student(const std::string& studentName, int roll, int numSub, const std::vector<std::string>& subNames)` : Overloaded constructor with basic student and subject information.
        - **Getter Methods:**
            - `getName() const` : Retrieves the student's name.
            - `getRollNumber() const` : Retrieves the student's roll number.
            - `getNumSubjects() const` : Retrieves the number of subjects.
            - `getSubjectNames() const` : Retrieves the names of subjects.
            - `getSubjectGrades() const` : Retrieves the grades for each subject.
        - **Setter Methods:**
            - `setName(const std::string& studentName)` : Sets the student's name.
            - `setRollNumber(int roll)` : Sets the student's roll number.
            - `setNumSubjects(int numSub)` : Sets the number of subjects.
            - `setSubjectNames(const std::vector<std::string>& subNames)` : Sets the names of subjects.
            - `setSubjectGrades(const std::vector<char>& subGrades)` : Sets the grades for each subject.
        - **Method:**
            - `calculateOverallGrade() const` : Calculates and returns the overall grade based on subject grades.

2. **Implement the `Student` Class (student.cpp):**

- Implement the methods declared in the `student.h` file in the corresponding source file (`student.cpp`).
- Provide logic for calculating the overall grade based on subject grades.
  - **Default Constructor:**
    - Initializes `name` to "Unknown" and sets default values for `rollNumber` and `numSubjects`.
  - **Overloaded Constructors:**
    - Overloaded constructors allow initialization of basic and subject-related information separately.
  - **Getter Methods:**
    - Retrieve basic student information (`getName()`, `getRollNumber()`).
    - Retrieve subject-related information (`getNumSubjects()`, `getSubjectNames()`, `getSubjectGrades()`).
  - **Setter Methods:**
    - Set basic student information (`setName()`, `setRollNumber()`).
    - Set subject-related information (`setNumSubjects()`, `setSubjectNames()`, `setSubjectGrades()`).
  - **Calculate Overall Grade Method:**
    - Calculates the overall grade based on subject grades.
    - Uses a mapping system for grades (A, B, C, D, F) based on average marks.

3. **Create the Main Program (main.cpp):**

    - Develop the main program (`main.cpp`) to showcase the functionality of the `Student` class.
    - Create instances of the `Student` class, set basic and subject-related information, and display the results.
    - Demonstrate the calculation of the overall grade.
        - **Include Statements:**
            - Includes the necessary header file `student.h` to use the `Student` class.
        - **Creating Students:**
            - Creates two instances of the `Student` class: `student1` using the default constructor and `student2` using the overloaded constructor.
        - **Setting Information:**
            - Uses setter methods to set basic student information (name, roll number) and subject-related information (number of subjects, subject names, subject grades) for both students.
        - **Displaying Information:**
            - Uses getter methods to display basic student information (name, roll number) and subject-related information (number of subjects, subject names, subject grades) for both students.
        - **Calculating and Displaying Overall Grade:**
            - Uses the `calculateOverallGrade()` method to calculate and display the overall grade for both students.

This `main.cpp` file serves as the entry point for the program and demonstrates the functionality of the extended Student Management System by creating, setting, displaying, and calculating grades for two students.

4. **Compile and Run:**

    - Compile the project using a C++ compiler (e.g., g++, Visual C++).
    - Run the executable to observe the output and verify that the program functions as expected.

Guidance:

- **Understanding Constructors:**
    - Ensure you understand the concept of constructors and how they are used in the `Student` class to initialize data members.
- **Data Encapsulation:**
    - Pay attention to the use of access specifiers to encapsulate data members appropriately, ensuring a clear distinction between public and private elements.
- **Calculating Overall Grade:**
    - Study the logic in the `calculateOverallGrade()` method to understand how the overall grade is calculated based on subject grades.
- **Demonstration:**
    - Use the program to create multiple instances of the `Student` class with different sets of information.
    - Display both basic and subject-related information for each student.

- Verify that the overall grade is calculated correctly.

This project will provide a comprehensive understanding of OOP principles through practical implementation. It encourages exploration and experimentation, fostering a deeper grasp of class design and functionality. Enjoy the learning journey!

Reference Implementation

student.h:

```
// student.h
#pragma once

#include <string>
#include <vector>

class Student {
private:
    // Data members for basic student information
    std::string name;
    int rollNumber;

    // Data members for subject-related information
    int numSubjects;
    std::vector<std::string> subjectNames;
    std::vector<char> subjectGrades;
```

```
public:
    // Default constructor
    Student();

    // Overloaded constructor with basic student information
    Student(const std::string& studentName, int roll);

    // Overloaded constructor with basic student and subject information
    Student(const std::string& studentName, int roll, int numSub, const s
td::vector<std::string>& subNames);

    // Getter methods for basic student information
    std::string getName() const;
    int getRollNumber() const;

    // Getter methods for subject-related information
    int getNumSubjects() const;
    std::vector<std::string> getSubjectNames() const;
    std::vector<char> getSubjectGrades() const;



    // Setter methods for basic student information
    void setName(const std::string& studentName);
    void setRollNumber(int roll);

    // Setter methods for subject-related information
    void setNumSubjects(int numSub);
    void setSubjectNames(const std::vector<std::string>& subNames);
    void setSubjectGrades(const std::vector<char>& subGrades);

    // Method to calculate overall grade based on subject grades
    char calculateOverallGrade() const;
};
```

student.cpp:

```cpp
// student.cpp
#include "student.h"

// Default constructor
Student::Student() : rollNumber(0), numSubjects(0) {


// Overloaded constructor with basic student and subject information
Student::Student(const std::string& studentName, int roll, int numSub, co
nst std::vector<std::string>& subNames)
    : rollNumber(roll), numSubjects(numSub), subjectNames(subNames) {
    name = studentName;
}

// Getter methods for basic student information
std::string Student::getName() const {
    return name;
}

int Student::getRollNumber() const {
    return rollNumber;
}

// Getter methods for subject-related information
int Student::getNumSubjects() const {
    return numSubjects;
}



std::vector<std::string> Student::getSubjectNames() const {
    return subjectNames;
}

std::vector<char> Student::getSubjectGrades() const {
    return subjectGrades;
}

// Setter methods for basic student information
void Student::setName(const std::string& studentName) {
    name = studentName;
}

void Student::setRollNumber(int roll) {
    rollNumber = roll;
}
```

```cpp
// Setter methods for subject-related information
void Student::setNumSubjects(int numSub) {
    numSubjects = numSub;
}

void Student::setSubjectNames(const std::vector<std::string>& subNames) {
    subjectNames = subNames;
}

void Student::setSubjectGrades(const std::vector<char>& subGrades) {
    subjectGrades = subGrades;
}



// Method to calculate overall grade based on subject grades
char Student::calculateOverallGrade() const {
    if (subjectGrades.empty()) {
        return 'N'; // Not enough data to calculate the overall grade
    }

    int totalMarks = 0;
    for (char grade : subjectGrades) {
        switch (grade) {
            case 'A':
                totalMarks += 90;
                break;
            case 'B':
                totalMarks += 80;
                break;
            case 'C':
                totalMarks += 70;
                break;
            case 'D':
                totalMarks += 60;
                break;
            case 'F':
                totalMarks += 50;
                break;
            default:
                totalMarks += 0;
        }
    }
```

```
        // Calculate the average and map it to a grade
        int averageMarks = totalMarks / subjectGrades.size();
        if (averageMarks >= 90) {
            return 'A';
        } else if (averageMarks >= 80) {
            return 'B';
        } else if (averageMarks >= 70) {
            return 'C';
        } else if (averageMarks >= 60) {
            return 'D';
        } else {
            return 'F';
        }
    }
```

main.cpp:

```
    // main.cpp
    #include "student.h"
    #include <iostream>

    int main() {
        // Create a student using the default constructor
        Student student1;

        // Set basic student information using setters
        student1.setName("Alice");
        student1.setRollNumber(101);

        // Set subject-related information using setters
        student1.setNumSubjects(3);
        student1.setSubjectNames({"Math", "Physics", "History"});
        student1.setSubjectGrades({'A', 'B', 'C'});
```

```
    // Display student information using getters
    std::cout << "Student Name: " << student1.getName() << std::endl;
    std::cout << "Roll Number: " << student1.getRollNumber() << std::end
l;

    // Display subject-related information using getters
    std::cout << "Number of Subjects: " << student1.getNumSubjects() << s
td::endl;
    std::cout << "Subject Names: ";
    for (const auto& subject : student1.getSubjectNames()) {
        std::cout << subject << " ";
    }
    std::cout << std::endl;

    std::cout << "Subject Grades: ";
    for (const auto& grade : student1.getSubjectGrades()) {
        std::cout << grade << " ";
    }
    std::cout << std::endl;



    // Calculate and display overall grade
    char overallGrade = student1.calculateOverallGrade();
    std::cout << "Overall Grade: " << overallGrade << std::endl;

    // Create another student using the overloaded constructor
    Student student2("Bob", 102, 2, {"Chemistry", "English"}, {'B',
'A'});

    // Display the second student's information
    std::cout << "\nStudent Name: " << student2.getName() << std::endl;
    std::cout << "Roll Number: " << student2.getRollNumber() << std::end
l;

    // Display subject-related information for the second student
    std::cout << "Number of Subjects: " << student2.getNumSubjects() << s
td::endl;
    std::cout << "Subject Names: ";
    for (const auto& subject : student2.getSubjectNames()) {
        std::cout << subject << " ";
    }
    std::cout << std::endl;
```

```
    std::cout << "Subject Grades: ";
    for (const auto& grade : student2.getSubjectGrades()) {
        std::cout << grade << " ";
    }
    std::cout << std::endl;

    // Calculate and display overall grade for the second student
    overallGrade = student2.calculateOverallGrade();
    std::cout << "Overall Grade: " << overallGrade << std::endl;

    return 0;
}
```

Reflection MCQs:

1. **What is the main purpose of access specifiers in C++?**

   - A. To define data members
   - B. To control the visibility and access of class members
   - C. To enable polymorphism
   - D. To set default values for members

   Correct Answer: B
2. **In C++, what does the `protected` access specifier indicate?**

   - A. Members are accessible only within the same class
   - B. Members are accessible from any class
   - C. Members are accessible only within the same class and its subclasses
   - D. Members are accessible only within the same file

   Correct Answer: C

3. **Why is it important to control access to class members using access specifiers?**

   - A. To reduce memory consumption
   - B. To improve code readability
   - C. To enhance security and prevent unauthorized access
   - D. To speed up program execution

   Correct Answer: C
4. **What is the default access specifier for members of a C++ class?**

   - A. public
   - B. private
   - C. protected
   - D. It depends on the compiler

5. **Which access specifier is commonly used for members that should not be directly accessed from outside the class or its subclasses?**

    - A. public
    - B. private
    - C. protected
    - D. internal

    Correct Answer: B
6. **Which of the following best describes the concept of data encapsulation in C++?**

    - A. Combining data and methods into a single unit
    - B. Hiding the implementation details of a class
    - C. Restricting access to certain members of a class
    - D. Allowing access to private members from external classes

    Correct Answer: B

Code Exercise:

Exercise 1:

- Lines 55-74 in `main.cpp` are commented bacuse they require another constructor oveload with all 5 parameters to initialize all 5 data members.
- Write the required constructor declaration in `student.h` and definition in `student.cpp` .
- Test your code if it works fine.

Exercise 2:

Implementing a Class for Employee Management

Objective: Create a C++ program to manage employee information using classes and object-oriented programming principles. This exercise focuses on class design, encapsulation, and access specifiers.

Requirements:

1. **Employee Class:**

    - Create a class named `Employee` with private data members for:
        - Employee ID (integer)
        - Employee Name (string)
        - Monthly Salary (float)

2. **Access Specifiers:**

- Implement access specifiers to control the visibility of data members. Use appropriate access specifiers to encapsulate the data.

3. **Constructors:**
   - Create a default constructor that initializes data members with default values.
   - Create an overloaded constructor that allows initializing all data members.
4. **Getter and Setter Methods:**
   - Implement getter methods to retrieve employee information.
   - Implement setter methods to set employee information.
5. **Salary Adjustment Method:**
   - Implement a method named `adjustSalary()` that takes a percentage as an argument and adjusts the monthly salary accordingly.
6. **Display Method:**
   - Implement a method named `displayDetails()` that displays all details of the employee.

Instructions:

1. **Class Definition:**
   - Define the `Employee` class in a header file (`employee.h`).
2. **Implementation:**
   - Implement the methods of the `Employee` class in a source file (`employee.cpp`).
3. **Main Program:**
   - Write a main program (`main.cpp`) to demonstrate the functionality of the `Employee` class.
   - Create instances of the `Employee` class, set and display employee details, and adjust salaries.
4. **Testing:**
   - Test the program with different employee instances and salary adjustments to ensure correct functionality.

Guidance:

- Use appropriate access specifiers to achieve data encapsulation.
- Ensure proper initialization of data members in constructors.
- Implement setter and getter methods to manipulate and retrieve data.
- Test the program thoroughly to validate the correctness of the implemented methods.

Example Starter Code (employee.h):

```cpp
// employee.h
#pragma once

#include <iostream>
#include <string>

class Employee {
private:
    int employeeID;
    std::string employeeName;
    float monthlySalary;

public:


    // Getter methods
    int getEmployeeID() const;
    std::string getEmployeeName() const;
    float getMonthlySalary() const;

    // Setter methods
    void setEmployeeID(int id);
    void setEmployeeName(const std::string& name);
    void setMonthlySalary(float salary);

    // Method to adjust monthly salary by a percentage
    void adjustSalary(float percentage);

    // Method to display employee details
    void displayDetails() const;
};
```

Example Starter Code (employee.cpp):

```cpp
// employee.cpp
#include "employee.h"

// Implement the methods of the Employee class here
// ...
```

Example Starter Code (main.cpp):