Object Oriented Programming (Spring2024)

Week01 (12/15/16-Feb-2024)

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

Definition and Principles of Object-Oriented Programming (OOP)

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable, self-contained objects. These objects represent real-world entities, encapsulating both data and the functions that operate on the data. OOP emphasizes the use of objects, classes, and the interactions between them to design and implement software.

Structured Programming Example:

In structured programming, the focus is on procedures and functions. Here's a C++ code example for calculating the area of a rectangle in a structured manner:

```cpp
#include <iostream>

// Function to calculate the area of a rectangle
double calculateRectangleArea(double length, double width) {
    return length * width;
}

int main() {
    // Input values
    double length, width;
    std::cout << "Enter length of the rectangle: ";
    std::cin >> length;
    std::cout << "Enter width of the rectangle: ";
    std::cin >> width;

    // Calculate and display area using the function
    double area = calculateRectangleArea(length, width);
    std::cout << "Area of the rectangle: " << area << std::endl;

    return 0;
}
```

Object-Oriented Programming (OOP) Example:

In OOP, the focus is on objects and classes. Here's a C++ code example for the same problem using an object-oriented approach:

```cpp
#include <iostream>

// Class definition for Rectangle
class Rectangle {
private:
    double length;
    double width;

public:
    // Constructor
    Rectangle(double l, double w) : length(l), width(w) {}

    // Member function to calculate the area
    double calculateArea() {
        return length * width;
    }
};



int main() {
    // Input values
    double length, width;
    std::cout << "Enter length of the rectangle: ";
    std::cin >> length;
    std::cout << "Enter width of the rectangle: ";
    std::cin >> width;

    // Create an object of the Rectangle class
    Rectangle rectangle(length, width);

    // Calculate and display area using the object
    double area = rectangle.calculateArea();
    std::cout << "Area of the rectangle: " << area << std::endl;

    return 0;
}
```

In this object-oriented programming example, the problem is approached by defining a `Rectangle` class. The class encapsulates both data members (`length` and `width`) and a member function (`calculateArea`). The program creates an object of the `Rectangle` class and uses it to calculate and display the area.

In summary, structured programming focuses on procedures and functions, while object-oriented programming emphasizes objects and classes. The OOP example uses encapsulation and a class to model the problem in a more modular and reusable manner

Procedural (Structured) Programming Focus:

1. **Focus on "How":**
   - **Explanation:** Procedural programming emphasizes the step-by-step procedure or sequence of operations to accomplish a task.
   - **Example:** In procedural code, you would explicitly outline the sequence of actions and operations needed to achieve the desired outcome. It often involves detailed control flow, specifying how each step is carried out.
2. **Implementation Details:**
   - **Example:** If you are calculating the area of a rectangle procedurally, you might explicitly describe the multiplication of length and width, possibly involving intermediate variables and step-by-step instructions.

Object-Oriented Programming (OOP) Focus:

1. **Focus on "What":**
   - **Explanation:** Object-oriented programming focuses on defining objects that encapsulate data and behavior, and it emphasizes what objects can do rather than how they do it.
   - **Example:** In OOP, you define a class representing a rectangle with a method to calculate its area. The emphasis is on what the object (rectangle) can do (calculate area) rather than the explicit steps involved.
2. **Abstraction and Encapsulation:**
   - **Example:** Instead of explicitly detailing the multiplication of length and width, an object-oriented solution might abstract this process. You create a `Rectangle` class, encapsulate the data (length and width) within it, and provide a method (`calculateArea()`) that hides the implementation details.

In the procedural example, the focus is on how to calculate the area, with explicit steps outlined in the `calculateRectangleArea` function. In the object-oriented example, the `Rectangle` class encapsulates the data and behavior related to a rectangle, and the emphasis is on what the object (rectangle) can do, abstracting away the implementation details.

This difference in focus reflects the shift from detailed procedural steps to a more abstract and modular representation of concepts in OOP. OOP promotes a higher level of abstraction and encapsulation, making the code more readable, maintainable, and adaptable to changes.

Doesn't OOP makes the solution complex?

Yes, in the given simple example of calculating the area of a rectangle, the object-oriented programming (OOP) solution might appear more complex compared to the structured programming solution. In small-scale programs or for simple problems, the additional structure introduced by classes and objects in OOP might seem like an overhead.

For simple tasks like calculating the area of a rectangle, the structured programming approach might indeed seem more concise and straightforward. The choice between structured and object-oriented programming often depends on the complexity of the problem and the overall design

Object-oriented programming (OOP) becomes advantageous in situations where the complexity of a software system increases, and certain design principles and features provided by OOP become more valuable. Here are scenarios where OOP is beneficial and how it provides advantages:

1. **Complexity and Scale:**
   - **Advantage:** OOP excels in managing complexity as the size and scale of a software project grow.
   - **How:** The encapsulation, abstraction, and modularization features of OOP help break down complex systems into manageable and understandable pieces. Classes and objects provide a way to structure code and separate concerns, making it easier to maintain and extend the system.
2. **Code Reusability:**
   - **Advantage:** OOP promotes code reuse, reducing redundancy and making development more efficient.
   - **How:** Through inheritance and polymorphism, common functionalities can be encapsulated in base classes, and derived classes can inherit and extend these functionalities. This minimizes code duplication and allows for modular and reusable code components.
3. **Modularity and Maintenance:**
   - **Advantage:** OOP supports modularity, making it easier to maintain and update software systems.
   - **How:** With well-defined classes and encapsulated functionalities, changes can be made to one part of the system without affecting others. This modular structure improves code maintenance, readability, and the ability to isolate and fix bugs or add new features.
4. **Real-World Modeling:**
   - **Advantage:** OOP is well-suited for modeling real-world entities and their interactions.
   - **How:** Classes and objects in OOP closely resemble real-world entities and their relationships. This makes it natural to express and design software solutions in a way that mirrors the structure and behavior of the problem domain.

In summary, the advantages of using OOP become apparent in large and complex software projects where modular design, code reusability, and maintainability are crucial. OOP provides a set of principles and features that promote a structured, organized, and scalable approach to software development. The choice to use OOP depends on the specific requirements and characteristics of the project at hand.

How does Object-Oriented Programming work?

1. **Objects and Classes:** In OOP, everything is treated as an object. Objects are instances of classes, which serve as blueprints for creating objects. A class defines the properties (attributes) and behaviors (methods) common to all objects of that type.
2. **Encapsulation in Action:** Encapsulation involves bundling data and methods that operate on the data within a class. This protects the internal state of an object and restricts direct access to its data. Accessors (getters) and mutators (setters) are used to control access to the object's attributes.
3. **Inheritance and Code Reuse:** Inheritance allows a new class to inherit properties and behaviors from an existing class. This promotes code reuse and the creation of a hierarchy of classes, with each level inheriting and extending the functionality of the previous level.
4. **Polymorphism and Flexibility:** Polymorphism enables a single interface to represent different types. It includes function overloading and overriding. Function overloading allows multiple functions with the same name but different parameters, while function overriding involves

Reflection Multiple-Choice Questions (MCQs):

1. **What is the primary purpose of Object-Oriented Programming (OOP)?**

   - A. Code obfuscation
   - B. Code reusability
   - C. Code duplication
   - D. Code compression

   Click to reveal the answer
2. **Which OOP principle involves bundling data and methods within a class?**

   - A. Inheritance
   - B. Polymorphism
   - C. Encapsulation
   - D. Abstraction

   Click to reveal the answer
3. **What does polymorphism in OOP allow for?**

   - A. Reusing code
   - B. Treating different objects as the same type
   - C. Creating hierarchical structures
   - D. Defining blueprints for objects

   Click to reveal the answer

Key Concepts: Encapsulation, Inheritance, Polymorphism

1. Encapsulation

What is Encapsulation?

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It involves bundling data (attributes) and methods (functions) that operate on the data into a single unit known as a class. This unit acts as a capsule, encapsulating the internal workings of an object and restricting direct access to its internal state.

Why Encapsulation?

1. **Data Protection:** Encapsulation protects the internal state of an object by restricting direct access to its attributes. Access to the data is controlled through methods, allowing the implementation of validation and ensuring data integrity.
2. **Abstraction:** Encapsulation provides a level of abstraction by hiding the complex implementation details of an object. Users interact with the object through its public interface, which consists of methods that expose the functionality without revealing the inner workings.

How Encapsulation Works:

1. **Private and Public Members:** In a class, attributes and methods can have different access specifiers. Private members are accessible only within the class, while public members are accessible from outside the class. This helps in controlling the visibility and access to the internal components of an object.
2. **Accessors (Getters) and Mutators (Setters):** To access and modify the private attributes of an object, accessor methods (getters) and mutator methods (setters) are used, respectively. These methods provide controlled access to the encapsulated data.

2. Inheritance

What is Inheritance?

Inheritance is a key concept in OOP that allows a new class (derived or child class) to inherit properties and behaviors from an existing class (base or parent class). This promotes code reuse and the creation of a hierarchical structure of classes.

Why Inheritance?

1. **Code Reusability:** Inheritance facilitates the reuse of code by allowing a new class to inherit the attributes and methods of an existing class. This reduces redundancy and promotes a more efficient and modular codebase.
2. **Hierarchy and Structure:** Inheritance supports the creation of a hierarchical structure of classes, where each level of the hierarchy inherits and extends the functionality of the previous level. This helps in organizing and managing complex systems.

How Inheritance Works:

1. **Base and Derived Classes:** The existing class is referred to as the base or parent class, and the new class is the derived or child class. The derived class inherits the attributes and methods of the base class and can also have additional members or override existing ones.
2. **Access Control:** Inheritance involves access specifiers (public, protected, private) that determine the visibility of the inherited members in the derived class. This controls how the derived class interacts with the inherited components.

3. Polymorphism

What is Polymorphism?

Polymorphism, meaning "many forms," is the ability of objects of different types to be treated as objects of a common base type. In OOP, polymorphism allows a single interface to represent different types of objects.

Why Polymorphism?

1. **Flexibility and Extensibility:** Polymorphism enhances flexibility by allowing a single interface to work with objects of various types. This promotes extensibility, enabling the addition of new classes without modifying existing code.
2. **Code Clarity:** Polymorphism improves code clarity by simplifying the interactions between objects. It allows developers to write more generic code that can be applied to a variety of object types.

How Polymorphism Works:

1. **Function Overloading:** Polymorphism includes function overloading, where multiple functions with the same name but different parameters can coexist. This allows a function to behave differently based on the input parameters.
2. **Function Overriding:** In polymorphism, function overriding occurs when a derived class provides a specific implementation for a function that is already defined in its base class. This allows a derived class to customize or extend the behavior of the inherited function.

Reflection Multiple-Choice Questions (MCQs):

1. **What does inheritance in OOP facilitate?**
   - A. Code obfuscation
   - B. Code reusability
   - C. Code duplication
   - D. Code compression
     Click to reveal the answer
2. **What is the core idea behind polymorphism?**
   - A. Code reusability
   - B. Flexibility and extensibility
   - C. Data protection
   - D. Hierarchy creation
     Click to reveal the answer

Brief Review of Basic C++ Syntax

What is Basic C++ Syntax?

Basic C++ syntax refers to the fundamental rules and structures used to write C++ programs. This review serves as a prerequisite for understanding and applying Object-Oriented Programming (OOP) principles in C++. Key aspects of basic C++ syntax include variables, data types, control structures, functions, and the use of libraries.

Why Review Basic C++ Syntax?

1. **Foundation for OOP:** A solid understanding of basic C++ syntax is crucial before delving into OOP. OOP builds upon the procedural programming concepts of C++, and a review ensures that all students have a common foundation.
2. **Prerequisite for Advanced Topics:** Many OOP concepts, such as classes, objects, and inheritance, heavily rely on the basic constructs of C++. A thorough review ensures that students are well-equipped to grasp the intricacies of OOP.

How to Review Basic C++ Syntax:

1. **Variables and Data Types:**
   - **What:** Variables store data, and data types define the type of data a variable can hold (e.g., int, float, char).
   - **Why:** Understanding variables and data types is essential for manipulating and storing information in C++ programs.
   - **How:** Discuss the declaration, initialization, and usage of variables with different data types.
2. **Control Structures (if, for, while):**
   - **What:** Control structures manage the flow of execution in a program (e.g., if statements for conditional execution, loops for iteration).
   - **Why:** These structures control the logic and order of operations, allowing for decision-making and repetition in programs.
   - **How:** Review the syntax and usage of if statements, for loops, and while loops. Provide examples to illustrate their application.

Example: Variables

```
#include <iostream>

int main() {
    // Basic data types
    int integerVar = 5;
    double doubleVar = 3.14;
    char charVar = 'A';

    // Output to console
    std::cout << "Integer: " << integerVar << std::endl;
    std::cout << "Double: " << doubleVar << std::endl;
    std::cout << "Character: " << charVar << std::endl;

    // User input
    int userInput;
    std::cout << "Enter an integer: ";
    std::cin >> userInput;
    std::cout << "You entered: " << userInput << std::endl;

    return 0;
}
```

Example: if statements

```cpp
#include <iostream>

int main() {
    // if statement
    int number = 10;
    if (number > 0) {
        std::cout << "Number is positive." << std::endl;
    } else if (number < 0) {
        std::cout << "Number is negative." << std::endl;
    } else {
        std::cout << "Number is zero." << std::endl;
    }

    // for loop
    for (int i = 1; i <= 5; ++i) {
        std::cout << "Iteration " << i << std::endl;
    }

    // while loop
    int countdown = 3;
    while (countdown > 0) {
        std::cout << "Countdown: " << countdown << std::endl;
        --countdown;
    }

    return 0;
}
```

3. **Functions:**

   - **What:** Functions encapsulate a set of instructions and can be called to perform a specific task.
   - **Why:** Functions promote code modularity and reusability by breaking down a program into manageable units.
   - **How:** Cover function declaration, definition, parameters, return types, and function calls. Illustrate the importance of functions in organizing code.

4. **Libraries:**

   - **What:** Libraries are collections of pre-written code that can be included in a program to extend its functionality.
   - **Why:** Using libraries enhances productivity by leveraging existing code and functionalities.
   - **How:** Introduce common C++ libraries, such as iostream for input/output and cmath for mathematical operations. Demonstrate how to include and use these libraries.

Example: Functions

```
#include <iostream>

// Function declaration
void greet() {
    std::cout << "Hello, world!" << std::endl;
}

// Function with parameters and return value
int add(int a, int b) {
    return a + b;
}

int main() {
    // Calling a function
    greet();

    // Using a function with parameters
    int sum = add(3, 4);
    std::cout << "Sum: " << sum << std::endl;

    return 0;
}
```

Reflection Multiple-Choice Questions (MCQs):

1. **What is the purpose of variables and data types in C++?**
    - A. To make programs more complex.
    - B. To store and manipulate data in programs.
    - C. To confuse programmers with unnecessary details.
    - D. To replace functions in programming.

   Click to reveal the answer
2. **How do functions contribute to code organization and reusability?**
    - A. By making programs longer and more complex.
    - B. By encapsulating a set of instructions and promoting modularity.
    - C. By avoiding the use of loops and conditional statements.
    - D. By preventing the inclusion of libraries in a program.

   Click to reveal the answer
3. **Why do programmers use libraries in C++ programs?**
    - A. To make programs more difficult to understand.
    - B. To create unnecessary dependencies.
    - C. To extend program functionality by leveraging pre-written code.
    - D. To replace basic C++ syntax.

Click to reveal the answer

Recap of Control Structures (if, for, while)

What are Control Structures?

Control structures in programming determine the flow of execution based on certain conditions or loops. In C++, three primary control structures are the `if` statement for conditional execution, `for` loop for iterative execution, and `while` loop for repetitive execution.

Why Recap Control Structures?

1. **Decision-Making with `if` Statements:**
    - **What:** The `if` statement allows for conditional execution based on a specified condition.
    - **Why:** `if` statements are essential for decision-making in programs, enabling different paths of execution based on varying conditions.
    - **How:** Recap the syntax of `if` statements, including the use of relational and logical operators. Provide examples illustrating their application.
2. **Iteration with `for` Loops:**
    - **What:** The `for` loop facilitates repetitive execution by defining an initialization, condition, and increment/decrement expression.
    - **Why:** `for` loops are crucial for iterating over a range of values or elements, reducing redundancy in code.
    - **How:** Review the structure of `for` loops and demonstrate their usage in scenarios such as iterating through arrays or performing numerical calculations.
3. **Repetition with `while` Loops:**
    - **What:** The `while` loop executes a block of code repeatedly as long as a specified condition is true.
    - **Why:** `while` loops are valuable for scenarios where the number of iterations is not known beforehand or when a condition must be met.
    - **How:** Revisit the syntax of `while` loops and provide examples showcasing their application, emphasizing the importance of updating loop control variables.

How to Recap Control Structures:

1. **`if` Statements:**
    - Discuss the syntax: `if (condition) { // code block } else if (condition) { // code block } else { // code block }`
    - Explain the role of logical operators (`&&` , `||` , `!`) in forming complex conditions.
    - Illustrate the use of nested `if` statements for handling multiple conditions.
2. **`for` Loops:**

- Introduce the structure: `for (initialization; condition; increment/decrement) { // code block }`
- Emphasize the importance of initialization, condition, and the increment/decrement expression.
- Demonstrate `for` loop applications, such as iterating through arrays or performing mathematical operations.

3. **`while` Loops:**

   - Recap the syntax: `while (condition) { // code block }`
   - Highlight the need for a proper update of loop control variables to avoid infinite loops.
   - Provide examples where `while` loops are suitable, such as input validation or reading data until a specific condition is met.

Reflection Multiple-Choice Questions (MCQs):

1. **What is the primary purpose of the `if` statement in programming?**

   - A. Facilitating repetitive execution.
   - B. Enabling decision-making based on conditions.
   - C. Initializing loop control variables.
   - D. Defining iterations over a range of values.

   Click to reveal the answer
2. **What is a key characteristic of the `for` loop in C++?**

   - A. It has no initialization step.
   - B. It does not use conditional expressions.
   - C. It is primarily used for decision-making.
   - D. It includes initialization, condition, and increment/decrement expressions.

   Click to reveal the answer

3. **When is the `while` loop particularly useful in programming?**

   - A. When a specific number of iterations is known beforehand.
   - B. When a loop should execute indefinitely.
   - C. When making decisions based on conditions.
   - D. When the loop control variables need not be updated.

   Click to reveal the answer
4. **What can help avoid infinite loops in `while` loops?**

   - A. Using the `for` loop instead.
   - B. Properly updating loop control variables.
   - C. Removing the loop condition.
   - D. Ignoring loop control variables.

   Click to reveal the answer

Understanding Functions in C++

What are Functions?

In C++, a function is a self-contained block of code that performs a specific task. Functions break down a program into smaller, manageable units, promoting modularity and reusability.

Why Understand Functions?

1. **Modularity:** Functions allow code to be organized into smaller, independent units. This modularity enhances code readability, maintenance, and reusability.
2. **Reusability:** Once defined, a function can be called multiple times from different parts of a program. This eliminates redundancy and promotes efficient code management.

How to Understand Functions:

1. **Function Declaration and Definition:**
    - **What:** A function declaration specifies the function's name, return type, and parameters. The definition includes the actual implementation of the function.
    - **Why:** Separating declaration and definition helps in clarifying the function's signature and its implementation.
    - **How:** Explain the syntax: `return_type function_name(parameters);` and provide examples of function declarations and definitions.
2. **Parameters and Arguments:**
    - **What:** Parameters are placeholders in the function declaration, while arguments are the actual values passed to the function.
    - **Why:** Parameters allow functions to accept input, making them versatile and applicable to various scenarios.
    - **How:** Demonstrate the use of parameters in functions, both in declaration and definition. Discuss different parameter types, including pass-by-value and pass-by-reference.
3. **Return Types:**
    - **What:** The return type specifies the data type of the value the function returns. A function with no return type is of type `void` .
    - **Why:** Return types define the kind of data a function produces, allowing for more precise data handling.
    - **How:** Illustrate different return types and the use of `void` when a function does not return a value. Show examples of functions returning values.
4. **Function Calls:**
    - **What:** Invoking a function in the program is known as a function call. It transfers control to the function, executes its code, and returns to the point of invocation.
    - **Why:** Function calls facilitate code organization, enabling the reuse of functionality.
    - **How:** Explain the syntax of a function call: `function_name(arguments);` and provide examples of calling functions in various scenarios.

Reflection Multiple-Choice Questions (MCQs):

1. **What is the purpose of separating function declaration and definition in C++?**
   - A. To confuse programmers.
   - B. To clarify the function's signature and implementation.
   - C. To avoid using functions altogether.
   - D. To increase redundancy in code.

   Click to reveal the answer
2. **What is the role of parameters in C++ functions?**
   - A. They define the return type of a function.
   - B. They specify the function's name.
   - C. They allow functions to accept input.
   - D. They handle function calls.

   Click to reveal the answer
3. **What does the return type of a C++ function indicate?**
   - A. The function's name.
   - B. The kind of data the function returns.
   - C. The function's parameters.
   - D. The absence of a return value.

   Click to reveal the answer
4. **Why are function calls important in programming?**
   - A. To confuse programmers.
   - B. To increase redundancy in code.
   - C. To facilitate code organization and reuse of functionality.
   - D. To eliminate the need for functions.

   Click to reveal the answer

Basics of Classes and Objects in C++

What are Classes and Objects?

Classes:

- **What:** A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.
- **Why:** Classes help structure code by grouping related data and functions together. They facilitate code organization, maintenance, and reusability.
- **How:** In C++, a class is declared using the `class` keyword, and it includes member variables (attributes) and member functions (methods).

Objects:

- **What:** An object is an instance of a class. It represents a real-world entity and is created based on the structure defined by the class.
- **Why:** Objects allow us to model and interact with entities in our programs in a way that mirrors real-world scenarios.

- **How:** Objects are created by instantiating a class. They encapsulate the data and provide a way to interact with the behaviors defined in the class.

How to Define and Use Classes in C++:

1. **Class Declaration:**
    - **What:** Declaring a class involves specifying its name, member variables, and member functions. The declaration is typically placed in a header file.
    - **Why:** A clear and organized class declaration ensures a well-defined structure for creating objects.
    - **How:** Demonstrate the syntax: `class ClassName { // members };` and provide examples of member variable and function declarations.
2. **Class Definition:**
    - **What:** The class definition provides the implementation details for the member functions declared in the class. It is usually placed in a separate source file.
    - **Why:** Separating the declaration from the definition improves code organization and readability.
    - **How:** Show the implementation of member functions using the scope resolution operator (`ClassName::`) and discuss encapsulation.

3. **Object Instantiation:**
    - **What:** Creating an object involves declaring a variable of the class type. This variable is an instance of the class, and it can be used to access the class's member variables and functions.
    - **Why:** Objects encapsulate data and behaviors, allowing for a more intuitive and modular design.
    - **How:** Illustrate the instantiation of objects using the syntax: `ClassName objectName;` and discuss how to access members using the dot operator (`objectName.member`).
4. **Access Control (Public and Private):**
    - **What:** Class members can be declared as public or private. Public members are accessible from outside the class, while private members are only accessible within the class.
    - **Why:** Access control restricts or allows external access to the internal details of a class, supporting encapsulation.
    - **How:** Explain the use of access specifiers (`public:` and `private:`) and demonstrate how they impact member accessibility.

Reflection Multiple-Choice Questions (MCQs):

1. **What is the primary purpose of a class in C++?**
    - A. To confuse programmers.
    - B. To structure code by grouping related data and functions.
    - C. To increase redundancy in code.

- D. To replace functions in programming.

Click to reveal the answer

2. **What does object instantiation involve in C++?**

    - A. Declaring member variables.
    - B. Creating a variable of the class type.
    - C. Defining member functions.
    - D. Accessing class members using the dot operator.

    Click to reveal the answer

3. **Why is access control important in C++ classes?**

    - A. To confuse programmers.
    - B. To increase redundancy in code.
    - C. To control external access to the internal details of a class.
    - D. To eliminate the need for classes altogether.

    Click to reveal the answer

4. **What is the purpose of the scope resolution operator (`::`) in C++ classes?**

    - A. To access class members using the dot operator.
    - B. To separate the declaration from the definition of a class.
    - C. To declare member variables and functions.
    - D. To provide implementation details for member functions.

    Click to reveal the answer

Declaration and Instantiation of Objects

What is Object Declaration and Instantiation?

Object Declaration:

- **What:** Object declaration involves specifying a variable of a certain class type. This variable is the placeholder for an instance of the class.
- **Why:** Declaring objects is essential for creating instances of classes, allowing for the encapsulation of data and behaviors.
- **How:** In C++, object declaration is done by specifying the class name followed by the object name.

Object Instantiation:

- **What:** Object instantiation is the process of creating an actual instance of a class. It involves allocating memory and initializing the object's member variables.
- **Why:** Instantiating objects enables the use of class functionalities, providing a means to interact with the encapsulated data and behaviors.
- **How:** Objects are instantiated using the class name followed by parentheses, similar to calling a function.

Why Declare and Instantiate Objects?

1. **Encapsulation:**
    - **Why:** Declaring and instantiating objects allows for the encapsulation of related data and behaviors within a single entity, promoting modularity and code organization.
2. **Reusability:**
    - **Why:** Objects can be reused throughout a program, eliminating the need to redefine the same data structures and functionalities. This enhances code reusability.
3. **Data Abstraction:**
    - **Why:** Objects abstract away the complex details of their internal implementation, providing a clear interface for interacting with the class.
4. **Code Organization:**
    - **Why:** Object-oriented programming emphasizes organizing code around real-world entities. Declaring and instantiating objects aligns with this organizational principle.

How to Declare and Instantiate Objects in C++:

1. **Object Declaration:**
    - **What:** Declare an object by specifying the class name followed by the object name.
    - **How:** `ClassName objectName;`
2. **Object Instantiation:**
    - **What:** Instantiate an object by using the class name followed by parentheses.
    - **How:** `ClassName objectName();`
3. **Accessing Class Members:**
    - **What:** After instantiation, members of the class, such as variables and functions, can be accessed using the dot operator (`objectName.member`).
    - **How:** Demonstrate accessing member variables and calling member functions of an object.
4. **Multiple Objects:**
    - **What:** Multiple objects of the same class can be declared and instantiated independently.
    - **How:** Declare and instantiate several objects of the same class to emphasize the independence of each instance.

Example: Class and Object

```cpp
#include <iostream>

// Class declaration
class Car {
public:
    // Data members
    std::string brand;
    int year;
    double price;

    // Member function
    void displayInfo() {
        std::cout << "Brand: " << brand << ", Year: " << year << ", Pric
e: $" << price << std::endl;
    }
};

int main() {
    // Creating an object of class Car
    Car myCar;

    // Accessing and modifying data members
    myCar.brand = "Toyota";
    myCar.year = 2022;
    myCar.price = 25000.0;

    // Calling a member function
    myCar.displayInfo();

    return 0;
}
```

Reflection Multiple-Choice Questions (MCQs):

1. **What is the purpose of declaring objects in C++?**
   - A. To confuse programmers.
   - B. To allocate memory for variables.
   - C. To create instances of classes and encapsulate related data and behaviors.
   - D. To eliminate the need for classes altogether.

   Click to reveal the answer
2. **Why is object instantiation important in OOP?**
   - A. To confuse programmers.
   - B. To abstract away the details of internal implementation.

- C. To increase redundancy in code.
- D. To emphasize procedural programming.

Click to reveal the answer

3. **How are class members accessed in C++ objects?**

    - A. Using parentheses.
    - B. Using the class name only.
    - C. Using the arrow operator (`->`).
    - D. Using the dot operator (`.`).

    Click to reveal the answer
4. **What is declaring and instantiating multiple objects of the same class is allowed in C++?**

    - A. To confuse programmers.
    - B. To create dependencies between objects.
    - C. To emphasize the independence of each instance.
    - D. To reduce code reusability.

    Click to reveal the answer

Setting Up C++ Environment on Windows Machine

Below is a step-by-step tutorial to set up Git Bash for compiling and running C++ programs on Windows using Sublime Text as the text editor.

Step 1: Install Git Bash

- **What:**
    - Git Bash provides a Unix-like command-line interface on Windows. It includes Git and a set of Unix commands.
- **How:**
    - Download Git Bash from the official website: Git for Windows (https://gitforwindows.org/).
    - Run the installer and follow the installation instructions.
    - During installation, select the option to include Git Bash in the system PATH.

Step 2: Install C++ Compiler (MinGW)

- **What:**
    - MinGW is a popular choice for a C++ compiler on Windows.
- **How:**
    - Open Git Bash.
    - Run the command to install MinGW:

```
pacman -S mingw-w64-x86_64-gcc
```

Step 3: Install Sublime Text

- **What:**

    - Sublime Text is a lightweight and versatile text editor.
- **How:**
    - Download and install Sublime Text from the official website: Sublime Text (https://www.sublimetext.com/).
    - Follow the installation instructions.

Step 4: Write and Save a C++ Program

- **What:**
    - Create a simple C++ program using Sublime Text.
- **How:**
    - Open Sublime Text.
    - Write a simple C++ program (e.g., `hello.cpp`):

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

    - Save the file with a `.cpp` extension.

Step 5: Compile and Run C++ Program using Git Bash

- **What:**
    - Use Git Bash commands to compile and run the C++ program.
- **How:**
    - Open Git Bash.
    - Navigate to the directory where your C++ file is located using the `cd` command:

```
cd path/to/your/cpp/file
```

    - Compile the C++ program using `g++` :

```
g++ hello.cpp -o hello.exe
```

    This command compiles the program and generates an executable named `hello.exe` .
    - Run the compiled executable:

```
./hello.exe
```

    This command executes the compiled program, and you should see the output.

Step 6: Repeat for Other Programs

- **What:**
    - Repeat the process for any additional C++ programs.

- **How:**
  - Write your C++ program.
  - Use Git Bash commands to compile and run as demonstrated in Step 4.

By following these steps, you've set up Git Bash to compile and run C++ programs on Windows using basic command-line tools. You can use any text editor to write your C++ code, and Git Bash to handle the compilation and execution.

Review Questions

1. **Question: What is Object-Oriented Programming (OOP)?**
   - **Answer:** Object-Oriented Programming (OOP) is a programming paradigm that uses objects, which are instances of classes, to organize code into modular and reusable structures.
2. **Question: Define encapsulation.**
   - **Answer:** Encapsulation is the bundling of data and methods that operate on the data within a class. It restricts direct access to some of the object's components, providing data hiding.
3. **Question: What are the three main principles of OOP?**
   - **Answer:** The three main principles of OOP are encapsulation, inheritance, and polymorphism.
4. **Question: Why is inheritance important in OOP?**
   - **Answer:** Inheritance allows a class to inherit properties and behaviors from another class. It promotes code reuse and establishes a relationship between classes.
5. **Question: What is polymorphism in OOP?**
   - **Answer:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables the same interface to represent different underlying forms.
6. **Question: Explain the concept of a class in C++.**
   - **Answer:** In C++, a class is a user-defined data type that represents a blueprint for creating objects. It defines data members and member functions that operate on the data.
7. **Question: What is the purpose of a constructor in C++?**
   - **Answer:** A constructor is a special member function in a class that is automatically called when an object is created. It is used to initialize the object's data members.
8. **Question: Differentiate between public and private access specifiers.**
   - **Answer:** Public members are accessible from outside the class, while private members are only accessible within the class. Encapsulation is achieved by making data members private.
9. **Question: How does C++ support the concept of function overloading?**
   - **Answer:** Function overloading in C++ allows multiple functions with the same name but different parameter lists. The compiler selects the appropriate function based on the context.
10. **Question: What is the purpose of the keyword 'this' in C++?**

- **Answer:** The 'this' pointer in C++ is used to refer to the current object. It is a pointer that points to the object for which the member function is invoked, enabling access to its members.

Code Exercise 1:

Execute the following code using any C++ compiler you prefer:

```cpp
#include <iostream>

class Rectangle {
public:
    double length;
    double width;

    // Method to calculate area
    double calculateArea() {
        return length * width;
    }
};

int main() {
    // Create an object of the Rectangle class
    Rectangle rectangle;

    double length, width;
    std::cout << "Enter length of the rectangle: ";
    std::cin >> length;
    std::cout << "Enter width of the rectangle: ";
    std::cin >> width;

    // Set dimensions directly using the object's public data members
    rectangle.length = length;
    rectangle.width = width;

    // Calculate and display area using the object
    double area = rectangle.calculateArea();
    std::cout << "Area of the rectangle: " << area << std::endl;

    return 0;
}
```

Code Exercise 2:

Implement an Enhanced Rectangle Class

Expand the functionality of the `Rectangle` class to include a check for whether the rectangle is a square or not. Modify the existing code to implement the following:

1. **Add a Function to Check if the Rectangle is a Square:**
   - Implement a member function within the `Rectangle` class named `isSquare` that returns a boolean indicating whether the rectangle is a square.
   - A rectangle is considered a square if its length and width are equal.
2. **Incorporate the New Functionality in the Main Program:**
   - In the `main` function, after setting the dimensions and calculating the area, use the `isSquare` function to check if the rectangle is a square.
   - Display an appropriate message based on the result.

Here's a skeleton code for reference:

```
#include <iostream>

class Rectangle {
public:
    double length;
    double width;

    // Method to calculate area
    double calculateArea() {
        return length * width;
    }

    /***TODO1: Method to check if the rectangle is a square***/







    }
};
```

```
int main() {
    // Create an object of the Rectangle class
    Rectangle rectangle;

    double length, width;
    std::cout << "Enter length of the rectangle: ";
    std::cin >> length;
    std::cout << "Enter width of the rectangle: ";
    std::cin >> width;

    // Set dimensions directly using the object's public data members
    rectangle.length = length;
    rectangle.width = width;

    // Calculate and display area using the object
    double area = rectangle.calculateArea();
    std::cout << "Area of the rectangle: " << area << std::endl;

    /***TODO2: Check if the rectangle is a square and display the result*
**/







    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required program**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind

Grading Strucutre:

- **Week3: Assessment 3%**
- **Week5: Assessment 7%**
- **Midterm: 30%**

- **Week11: Assessment 3%**
- **Week13: Assessment 7%**
- **Project: 15%**
- **Final: 35%**

In [ ]:

```
```