l1-introduction-to-OOP

July 22, 2024

1 Lecture notes on OOP (C++)

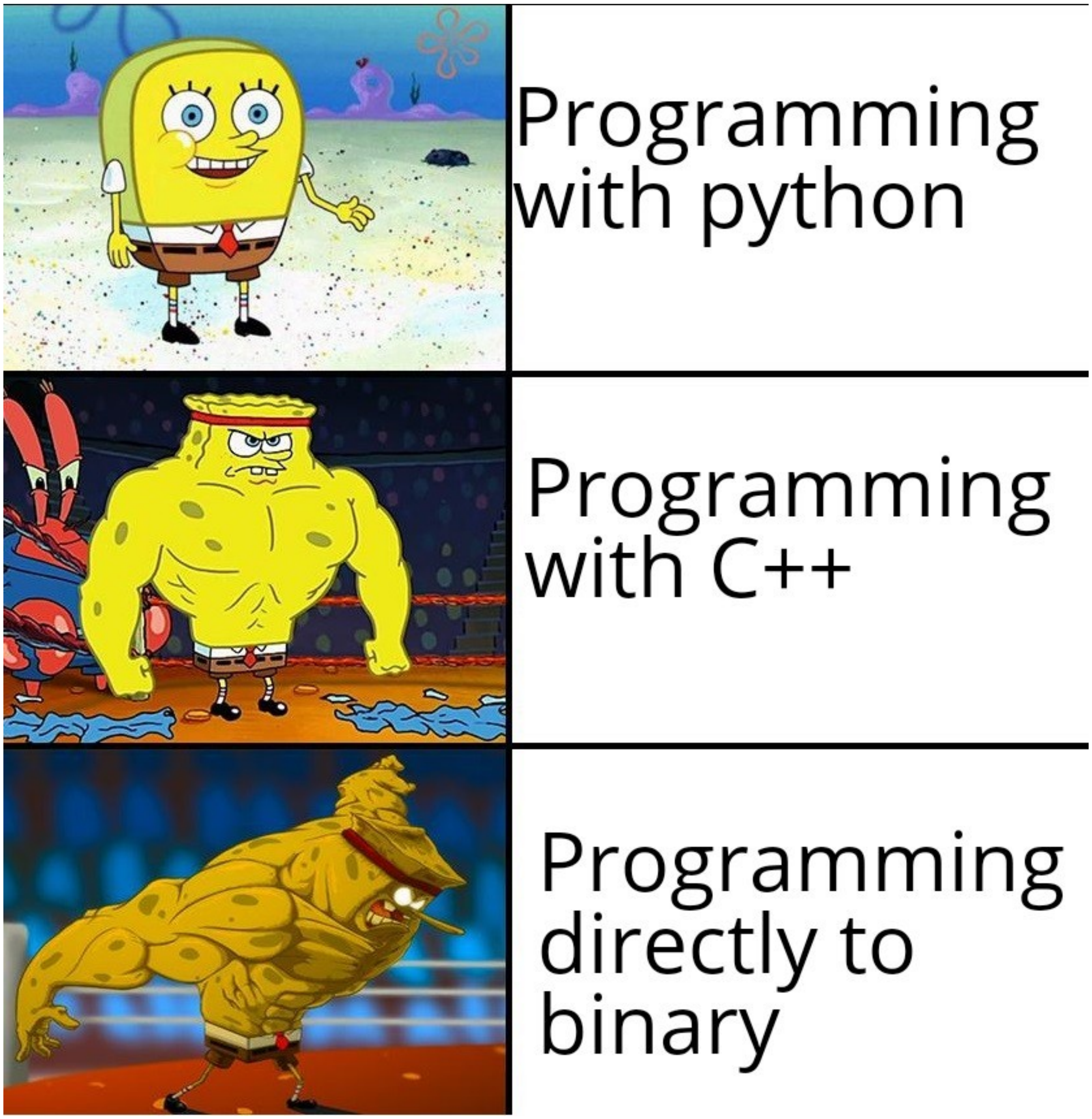
By Jason Pandian *Assistant Professor, Department of Information Technology*

2 INTRODUCTION TO OOP

Object Oriented Programming concepts - Features – Benefits of Object Oriented Methodology-Input and Output statements - Decision control and looping statements-Functions-Arrays-Classes and Objects-Memory allocation - Array of objects – Constructors - Destructors

3 Prerequestories

- Common Sense
- Little Logics
- Simple math Operations
- Focus # - **Forgot Python**



4 From C to C with Class

- C++ is a powerful general-purpose programming language that has significantly influenced the world of software development.
- It was created by Bjarne Stroustrup at Bell Labs starting in 1979 as an extension of the C programming language.
- The primary aim was to add object-oriented features to the C language while maintaining its efficiency and flexibility.

5 Need For Compilers

!gcc –version

[7]:
```
!gcc --version
```

```
gcc (Debian 12.2.0-14) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

[1]:
```
%%writefile unit1/simcls.cpp

#include <iostream>  // Required for std::cout
#include <string>    // Required for std::string

class Car {
public:
    std::string brand;   // Data member (attribute)
    std::string model;   // Data member (attribute)
    int year;            // Data member (attribute)

    void printDetails() {   // Member function (method)
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: "␣
 ↪<< year << std::endl;
    }
};

int main() {
    Car myCar;                 // Creating an object of the Car class
    myCar.brand = "Toyota";    // Setting the brand of myCar
    myCar.model = "Corolla";   // Setting the model of myCar
    myCar.year = 2020;         // Setting the year of myCar
    myCar.printDetails();      // Calling the printDetails method

    Car myCar1;                 // Creating an object of the Car class
    myCar1.brand = "Maruthi";    // Setting the brand of myCar
    myCar1.model = "SWIFT";   // Setting the model of myCar
    myCar1.year = 2024;         // Setting the year of myCar
    myCar1.printDetails();      // Calling the printDetails method

    return 0;
}
```

```
Overwriting unit1/simcls.cpp
```

[5]:
```
!gcc unit1/simcls.cpp -lstdc++ -o unit1/simcls -std=c++11

# Run the compiled program
!./unit1/simcls
```

```
Brand: Toyota, Model: Corolla, Year: 2020
Brand: Maruthi, Model: SWIFT, Year: 2024
```

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code to manipulate that data. It allows for more organized, modular, and reusable code. Here's an introduction to the core concepts of OOP in C++:

5.1 1. Classes and Objects

Class: A class is a blueprint for creating objects. It defines a type of object by bundling data and methods that operate on the data.

Object: An object is an instance of a class. It represents a specific entity with its own state and behavior.

[28]:
```
%%writefile unit1/simpleclass.cpp



#include <iostream>
using namespace std;

class Car {
public:
    // Data members
    string brand;
    int year;

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

int main() {
    // Create an object of the Car class
    Car myCar;
    myCar.brand = "Toyota";
    myCar.year = 2022;
    myCar.displayInfo(); // Output: Brand: Toyota, Year: 2022
    return 0;
}
```

```
Overwriting unit1/simpleclass.cpp
```

[29]:
```
!gcc unit1/simpleclass.cpp -lstdc++ -o unit1/simpleclass -std=c++11

# Run the compiled program
!./unit1/simpleclass
```

```
Brand: Toyota, Year: 2022
```

5.2 2. Encapsulation

Encapsulation: The concept of wrapping data (attributes) and methods (functions) into a single unit called a class. It restricts direct access to some of the object's components.

[10]:
```
%%writefile unit1/encap.cpp

#include <iostream> // Include the input-output stream library
using namespace std; // Use the standard namespace

// Define the BankAccount class
class BankAccount {
private:
    double balance; // Private member variable to store the account balance

public:
    // Constructor to initialize the balance with the provided initial balance
    BankAccount(double initialBalance) : balance(initialBalance) {}

    // Member function to deposit money into the account
    void deposit(double amount) {
        if (amount > 0) { // Check if the deposit amount is positive
            balance += amount; // Add the deposit amount to the balance
        }
    }

    // Member function to withdraw money from the account
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) { // Check if the withdrawal␣
↪amount is positive and less than or equal to the balance
            balance -= amount; // Subtract the withdrawal amount from the␣
↪balance
        }
    }

    // Member function to check the current balance of the account
    double getBalance() {
        return balance; // Return the current balance
    }
};

// Main function to demonstrate the usage of the BankAccount class
int main() {
    BankAccount myAccount(1000); // Create a BankAccount object with an initial␣
↪balance of 1000
    cout << "Current balance: " << myAccount.getBalance() << endl; // Output␣
↪the current balance
```

```
    myAccount.deposit(500); // Deposit 500 into the account
    cout << "Current balance: " << myAccount.getBalance() << endl; // Output␣
↪the current balance

    myAccount.withdraw(200); // Withdraw 200 from the account
    cout << "Current balance: " << myAccount.getBalance() << endl; // Output␣
↪the current balance
    return 0; // Return 0 to indicate successful execution
}
```

```
Overwriting unit1/encap.cpp
```

[11]:
```
!gcc unit1/encap.cpp -lstdc++ -o unit1/encap -std=c++11

# Run the compiled program
!./unit1/encap
```

```
Current balance: 1000
Current balance: 1500
Current balance: 1300
```

5.3 3. Inheritance

Inheritance: A mechanism where a new class (derived class) inherits properties and behaviors from an existing class (base class). This helps to promote code reusability.

[47]:
```
%%writefile unit1/inhert.cpp

#include <iostream> // Include the input-output stream library
using namespace std; // Use the standard namespace

// Define the Vehicle class
class Vehicle {
public:
    // Member function to start the vehicle
    void start() {
        cout << "Vehicle starting..." << endl;
    }

    void stop() {
        cout << "Vehicle stopped..." << endl;
    }
};

// Define the Car class, which inherits from the Vehicle class
class Car : public Vehicle {
public:
    // Member function to drive the car
```

```
    void drive() {
        cout << "Car is driving." << endl;
    }
};

// Main function to demonstrate the usage of the Vehicle and Car classes
int main() {
    Car myCar; // Create a Car object named myCar
    myCar.start(); // Call the inherited start method from the Vehicle class on␣
↪the myCar object
    myCar.drive(); // Call the drive method from the Car class on the myCar␣
↪object
    myCar.stop(); // Call the inherited stop method from the Vehicle class on␣
↪the myCar object
    return 0; // Return 0 to indicate successful execution
}
```

Overwriting unit1/inhert.cpp

[46]:
```
!gcc unit1/inhert.cpp -lstdc++ -o unit1/inhert -std=c++11

# Run the compiled program
!./unit1/inhert
```

Vehicle starting…
Car is driving.
Vehicle stopped…

5.4 4. Polymorphism

Polymorphism: The ability to present the same interface for different underlying data types. It allows one function or operator to operate in different ways depending on the context.

[53]:
```
%%writefile unit1/poly.cpp
// Directive to write this code to a file named poly.cpp in the unit1 directory

#include <iostream> // Include the input-output stream library
using namespace std; // Use the standard namespace

// Define the Shape class
class Shape {
public:
    // Virtual member function to draw a shape
    virtual void draw() {
        cout << "Drawing a shape." << endl;
    }
};

```

```
// Define the Circle class, which inherits from the Shape class
class Circle : public Shape {
public:
    // Override the draw function to draw a circle
    void draw() override {
        cout << "Drawing a circle." << endl;
    }
};

// Define the Square class, which inherits from the Shape class
class Square : public Shape {
public:
    // Override the draw function to draw a square
    void draw() override {
        cout << "Drawing a square." << endl;
    }
};

// Main function to demonstrate polymorphism with Shape, Circle, and Square␣
 ↪classes
int main() {
    Shape* shapePtr; // Declare a pointer to Shape

    Circle circle; // Create a Circle object
    Square square; // Create a Square object

    shapePtr = &circle; // Point shapePtr to the Circle object
    shapePtr->draw(); // Call the draw function through the Shape pointer,␣
 ↪outputs "Drawing a circle."

    shapePtr = &square; // Point shapePtr to the Square object
    shapePtr->draw(); // Call the draw function through the Shape pointer,␣
 ↪outputs "Drawing a square."

    return 0; // Return 0 to indicate successful execution
}
```

```
Overwriting unit1/poly.cpp
```

[52]:
```
!gcc unit1/poly.cpp -lstdc++ -o unit1/poly -std=c++11

# Run the compiled program
!./unit1/poly
```

```
Drawing a circle.
Drawing a square.
```

5.5 5. Abstraction

Abstraction: The concept of hiding the complex implementation details and showing only the essential features of an object. It allows focusing on what an object does instead of how it does it.

[54]:
```
%%writefile unit1/abst.cpp


#include <iostream> // Include the input-output stream library
using namespace std; // Use the standard namespace

// Define the AbstractShape class
class AbstractShape {
public:
    // Pure virtual function to draw a shape
    // This makes AbstractShape an abstract class
    virtual void draw() = 0;
    // = 0 indicates that this function is pure virtual
};

// Define the Rectangle class, which inherits from the AbstractShape class
class Rectangle : public AbstractShape {
public:
    // Override the pure virtual function to draw a rectangle
    void draw() override {
        cout << "Drawing a rectangle." << endl;
    }
};

// Main function to demonstrate the usage of the AbstractShape and Rectangle␣
↪classes
int main() {
    Rectangle rect; // Create a Rectangle object
    rect.draw(); // Call the draw function, outputs "Drawing a rectangle."
    return 0; // Return 0 to indicate successful execution
}
```

```
Writing unit1/abst.cpp
```

[55]:
```
!gcc unit1/abst.cpp -lstdc++ -o unit1/abst -std=c++11

# Run the compiled program
!./unit1/abst
```

```
Drawing a rectangle.
```

5.6 Quick Summary

- Classes and Objects: Define and create entities with state and behavior.

- Encapsulation: Hide the internal state and only expose necessary functionality.
- Inheritance: Create new classes based on existing classes to promote reuse.
- Polymorphism: Use the same function name for different implementations.
- Abstraction: Focus on essential qualities of objects and ignore irrelevant details.

6 Object-Oriented Methodology

Object-Oriented Methodology (OOM) offers several features and benefits that make it a popular paradigm for designing and developing software systems. Here's a detailed look at both:

6.1 Features of Object-Oriented Methodology

1. Encapsulation is the bundling of data and methods that operate on the data into a single unit or class. It restricts access to some of the object's components. - Implementation: Access specifiers like private, protected, and public control the visibility of class members.

2. Inheritance allows a new class (derived class) to inherit properties and behaviors from an existing class (base class). - Types: Single, multiple, and hierarchical inheritance.

3. Polymorphism allows methods to do different things based on the object it is acting upon. It provides a way to perform the same operation in different ways. - Types: Compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

4. Abstraction involves hiding the complex implementation details and showing only the essential features of the object. It helps in focusing on what an object does rather than how it does it. - Implementation: Abstract classes and interfaces define abstract methods without providing a concrete implementation.

5. Class and Object: Classes are blueprints for creating objects. Objects are instances of classes that encapsulate both data and methods. - Characteristics: Objects have state (attributes) and behavior (methods).

6.2 Benefits of Object-Oriented Methodology

1. Modularity: OOM promotes modularity by organizing code into classes. Each class can be developed, tested, and debugged independently. - Benefit: Makes it easier to manage and understand complex software systems.

2. Reusability: Classes and objects can be reused across different programs. Inheritance allows new classes to reuse code from existing classes. - Benefit: Reduces redundancy and improves code maintenance.

3. Maintainability: Encapsulation and modularity make it easier to modify and extend existing code. Changes in one part of the system can be made with minimal impact on other parts. - Benefit: Simplifies debugging and updates, leading to more robust software.

4. Scalability: OOM supports scalable design by allowing the addition of new classes and objects. It facilitates the development of large and complex systems. - Benefit: Enhances the ability to build and manage large-scale applications.

5. Flexibility: Polymorphism allows objects to be treated as instances of their parent class, enabling flexible and dynamic method execution. - Benefit: Provides flexibility in code implementation and facilitates future enhancements.

6. Abstraction: By focusing on high-level operations rather than low-level details, OOM makes complex systems easier to understand and manage. - Benefit: Improves code readability and reduces complexity.

7. Code Reusability: Inheritance and composition enable the reuse of existing code to create new functionality. - Benefit: Saves development time and effort by leveraging existing, tested code.

8. Improved Collaboration: OOM allows different team members to work on different classes simultaneously. Classes can be designed and tested independently. - Benefit: Facilitates collaborative development and parallel work.

9. Real-World Modeling: OOM models real-world entities as objects, which can make it easier to understand and design systems based on real-world scenarios. - Benefit: Enhances the alignment of software design with real-world requirements and logic.

1**0. Error Reduction:** Encapsulation hides internal data and methods, reducing the likelihood of unintended interference and errors. - Benefit: Improves code reliability and stability.

6.3 Quick Summary

Object-Oriented Methodology enhances software development through its features of encapsulation, inheritance, polymorphism, and abstraction. These features provide numerous benefits, including modularity, reusability, maintainability, scalability, flexibility, and improved collaboration. By modeling real-world entities and relationships, OOM helps in building robust, maintainable, and scalable software systems.

Text Books

1. ReemaThareja, ”Object Oriented Programming with C++”,Third Edition, Oxford University Press,New Delhi,2018 (UNIT 1)
    2. Herbert Schildt, “Java: The Complete Reference”, 12th Edition, McGraw Hill Education, New Delhi, 2021.(UNIT 2 to 5)

7 Any Questions or Doubts?

Refer the Lectures/Tutorials GitHub Page

[ ]:

```

```