# **Unit-1: Principles of OOP**

## 1. What is Object-Oriented Programming (OOP)?

 Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software programs. It allows for the modeling of real-world entities and their interactions, promoting code reusability and modularity.

## 2. How does procedural programming differ from object-oriented programming?

 Procedural programming focuses on functions and sequences of steps to perform tasks, whereas object-oriented programming organizes software design around data, or objects, and their interactions. OOP supports encapsulation, inheritance, and polymorphism, which are not inherent in procedural programming.

## 3. What are the basic concepts of object-oriented programming?

- The basic concepts of OOP are:
  - Encapsulation: Bundling data with the methods that operate on it.
  - Abstraction: Hiding complex implementation details and showing only the necessary features.
  - Inheritance: Creating new classes from existing ones to promote code reuse.
  - **Polymorphism:** Allowing objects to be treated as instances of their parent class.

## 4. Can you name some object-oriented programming languages?

• Some OOP languages include C++, Java, Python, C#, Ruby, and Swift.

## 5. What are the benefits of using OOP over procedural programming?

- Benefits of OOP include:
  - Enhanced modularity and code organization.
  - Easier maintenance and modification of code.
  - Reusability of code through inheritance.
  - Flexibility through polymorphism.

## 6. Explain the concept of encapsulation with an example.

 Encapsulation is the practice of bundling data and methods that work on that data within a single unit, or class, and restricting access to some of the object's components. For example, in a class Car, private variables like speed and fuelLevel can only be accessed and modified through public methods like
accelerate() and refuel().

# Unit-2: Basics of C++

## 1. How does C++ differ from C?

• C++ is an extension of C that includes object-oriented features such as classes and inheritance. C++ supports function overloading, operator overloading, and has better type checking compared to C.

# 2. Write a simple C++ program to print "Hello, World!" and explain the steps involved in compiling and linking.

```
cpp
#include <iostream>
using namespace std;
int main() {
   cout << "Hello, World!" << endl;
   return 0;
}</pre>
```

- Compiling: The source code is converted into object code by the compiler.
- Linking: The object code is linked with libraries to produce an executable file.

#### 3. What are tokens in C++? Provide examples.

Tokens are the smallest units in a C++ program and include keywords (e.g., int, return), identifiers (e.g., variable names), literals (e.g., 10, "hello"), operators (e.g., +, -), and punctuation (e.g., {, }).

## 4. What is a reference variable in C++? How is it different from a pointer?

A reference variable is an alias for another variable, created using the & operator. Unlike pointers, references cannot be null, do not need dereferencing to access the value, and must be initialized when declared.

#### 5. List some keywords in C++ and explain their use.

Keywords include int, float, if, else, while, for, class, return, etc.
 They have predefined meanings and are used to perform specific operations in the program.

## 6. Explain the different data types available in C++.

- C++ data types include:
  - **Primitive types:** int, char, float, double, bool.
  - Derived types: Arrays, pointers, references.
  - User-defined types: Classes, structures, unions, enums.

## **Unit-3: Operators and Type Casting**

- 1. What are the different types of operators in C++?
  - Operators include arithmetic (+, -, \*, /), relational (==, !=, <, >), logical (&&, ||, !), bitwise (&, |, ^), assignment (=, +=, -=), and others like increment (++), decrement (--).

## 2. Explain the scope resolution operator with an example.

• The scope resolution operator : : is used to access global variables or class members. For example:

```
cpp
int x = 10; // Global variable
class MyClass {
public:
    int x;
    void setX(int x) {
        this->x = x; // Using this pointer
    }
};
int main() {
    MyClass obj;
    obj.setX(20);
    cout << ::x << " " << obj.x; // Accessing global x and</pre>
class member x
    return 0;
}
```

3. How do member dereferencing operators work in C++?

 Member dereferencing operators, like . (dot) for accessing class members through an object and -> (arrow) for accessing class members through a pointer to an object, are used to refer to members of objects and pointers.

## 4. What is type casting? Differentiate between implicit and explicit type casting.

• Type casting converts a variable from one type to another. Implicit type casting is done automatically by the compiler, while explicit type casting requires the programmer to specify the conversion.

```
cpp
// Implicit
int a = 10;
double b = a;
// Explicit
double c = 9.8;
int d = (int)c;
```

5. Provide an example of implicit type casting in C++.

```
cpp
int num = 10;
double result = num; // Implicitly casts int to double
```

6. Provide an example of explicit type casting in C++.

```
cpp
double value = 9.5;
int intValue = (int)value; // Explicitly casts double to int
```

## **Unit-4: Control Structures**

- 1. What are decision-making controls in C++? Provide examples.
  - Decision-making controls include if, else if, else, switch. They execute code based on certain conditions.

срр

```
int num = 10;
if (num > 0) {
    cout << "Positive";
} else {
    cout << "Non-positive";
}
```

## 2. Explain the use of iterative controls in C++ with an example.

• Iterative controls include for, while, do-while loops. They are used for repetitive execution of code.

```
cpp
for (int i = 0; i < 5; i++) {
    cout << i << " ";
}</pre>
```

## 3. What are jumping controls in C++? Provide examples.

• Jumping controls include break, continue, goto. They alter the flow of control in loops and other structures.

```
cpp
for (int i = 0; i < 10; i++) {
    if (i == 5) break; // Exits the loop when i is 5
    cout << i << " ";
}</pre>
```

#### 4. How does the 'if' statement work in C++?

• The if statement evaluates a condition and executes the associated block of code if the condition is true.

```
cpp
int x = 5;
if (x > 0) {
    cout << "Positive number";
}</pre>
```

#### 5. Explain the 'switch-case' statement with an example.

• The switch-case statement executes one case from multiple possible values of an expression.

```
cpp
int day = 2;
switch (day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    default: cout << "Invalid day"; break;
}
```

#### 6. Compare and contrast 'while' and 'do-while' loops in C++.

• The while loop checks the condition before executing the loop body, whereas the do-while loop checks the condition after executing the loop body at least once.

```
// while loop
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
// do-while loop
int j = 0;
do {
    cout << j << " ";
    j++;
} while (j < 5);</pre>
```

# **Unit-5: Pointers and Structures**

срр

1. What is a function prototype? Why is it important in C++?

 A function prototype declares a function's name, return type, and parameters without its body. It helps the compiler understand how to call the function before its actual definition.

```
cpp
void display(); // Function prototype
int main() {
    display();
    return 0;
}
void display() {
    cout << "Hello, World!";
}
```

#### 2. How are pointers used in C++? Provide an example.

 Pointers store the address of another variable and are used for dynamic memory allocation, array handling, and more.

```
cpp
int var = 10;
int *ptr = &var; // Pointer to var
cout << *ptr; // Output: 10</pre>
```

#### 3. What are the limitations of C structures?

C structures cannot have member functions, access specifiers, or inheritance.
 They are primarily used for grouping data, not behavior.

#### 4. Explain the difference between structures in C and C++.

- C++ structures can have member functions, access specifiers, and inheritance, making them more similar to classes. In C, structures are purely data containers.
- 5. Write a simple program using pointers in C++.

```
cpp
int main() {
    int num = 10;
    int *ptr = #
```

```
cout << "Value: " << *ptr << ", Address: " << ptr;
return 0;
}
```

## 6. What is the main function in C++ and why is it important?

• The main function is the entry point of a C++ program. The execution of the program starts and ends with the main function.

```
cpp
int main() {
   cout << "Hello, World!";
   return 0;
}</pre>
```

# **Unit-6: Classes and Objects**

- 1. What is a class in C++? How is it defined?
  - A class is a blueprint for creating objects, encapsulating data and methods. It is defined using the class keyword.

```
cpp
class Car {
public:
    string model;
    void display() {
        cout << "Model: " << model;
    }
};</pre>
```

2. Provide a sample C++ program that uses a class and explain its components.

```
cpp
class Car {
  public:
    string model;
    void display() {
        cout << "Model: " << model;</pre>
```

```
};
int main() {
   Car car1;
   car1.model = "Tesla";
   car1.display();
   return 0;
}
```

Components: Class definition (Car), class members (model, display), object
 creation (car1), and member access (car1.model, car1.display()).

## 3. What are access specifiers in C++? List and explain them.

- Access specifiers control the visibility of class members:
  - **public:** Members are accessible from outside the class.
  - **private:** Members are accessible only within the class.
  - **protected:** Members are accessible within the class and its derived classes.

#### 4. How do you define member functions in C++?

• Member functions can be defined inside the class or outside using the scope resolution operator.

```
cpp
class Car {
public:
    void display();
};
void Car::display() {
    cout << "Car display function";
}</pre>
```

#### 5. What is the significance of nesting member functions in C++?

Nesting member functions means defining a function within another function.
 It can improve code organization and readability.

срр

```
class Example {
public:
    void outerFunction() {
        innerFunction();
    }
private:
    void innerFunction() {
        cout << "Inner function";
    }
};</pre>
```

- 6. Explain the difference between a class and an object.
  - A class is a blueprint or template for creating objects. An object is an instance of a class, containing data and methods defined by the class.

## **Unit-7: More on Classes and Objects**

- 1. How do you define functions inside a class in C++?
  - Functions can be defined inside the class definition itself.

```
cpp
class Car {
public:
    void display() {
        cout << "Display function inside class";
    }
};</pre>
```

#### 2. What are private member functions and how are they used?

• Private member functions are accessible only within the class and are used to encapsulate helper functions.

```
cpp
class Example {
public:
    void publicFunction() {
        privateFunction();
    }
```

```
private:
    void privateFunction() {
        cout << "Private function";
    }
};
```

#### 3. Explain how arrays can be used within a class in C++.

 Arrays can be class members and used to store multiple values of the same type.

```
cpp
class Student {
  public:
    int scores[5];
    void displayScores() {
       for (int i = 0; i < 5; i++) {
           cout << scores[i] << " ";
        }
    }
};
```

## 4. Describe how memory is allocated for objects in C++.

 Memory for objects is allocated either on the stack (for local objects) or on the heap (for dynamically allocated objects using new).

```
cpp
Student s1; // Stack allocation
Student *s2 = new Student(); // Heap allocation
```

- 5. What is the difference between defining a function inside the class and outside the class?
  - Defining a function inside the class makes it inline, potentially increasing execution speed. Defining it outside the class separates the interface from implementation, improving readability and maintainability.
- 6. Provide an example of a class with private member functions.

```
class Example {
public:
    void publicFunction() {
        privateFunction();
    }
private:
    void privateFunction() {
        cout << "Private function";
    }
};</pre>
```

## **Unit-8: Handling Functions**

- 1. Explain the different function calling mechanisms in C++: call by value, call by address, and call by reference.
  - **Call by Value:** Copies the actual value of an argument into the formal parameter.
  - **Call by Address:** Passes the address of an argument into the formal parameter.
  - Call by Reference: Passes a reference to the actual argument.
- 2. How are objects passed as function arguments in C++?
  - Objects can be passed by value, by pointer, or by reference.

срр

```
void display(Student s); // By value
void display(Student *s); // By pointer
void display(Student &s); // By reference
```

3. Provide an example of a function that uses call by value.

```
cpp
void display(int num) {
    cout << num;
}
int main() {
```

```
int a = 10;
display(a);
return 0;
```

}

4. Provide an example of a function that uses call by reference.

```
cpp
void display(int &num) {
    cout << num;
}
int main() {
    int a = 10;
    display(a);
    return 0;
}
```

## 5. What are the advantages of using call by reference over call by value?

• Call by reference avoids copying large data structures, reduces memory usage, and allows the function to modify the original variable.

## 6. Explain the concept of function overloading in C++.

 Function overloading allows multiple functions with the same name but different parameter lists. It enables the same function to handle different data types or numbers of arguments.

## **Unit-9: More on Functions**

## 1. What are inline functions? How are they different from regular functions?

• Inline functions are expanded in line where they are called, reducing function call overhead. They are defined using the inline keyword.

```
cpp
inline void display() {
    cout << "Inline function";
}</pre>
```

#### 2. How do you make an outside function inline in C++?

• By using the inline keyword in both the declaration and definition.

```
cpp
class Example {
  public:
     inline void display();
};
inline void Example::display() {
     cout << "Inline function";
}</pre>
```

#### 3. What are friend functions and how are they used in C++?

• Friend functions are non-member functions that can access private and protected members of a class. They are declared using the friend keyword.

```
cpp
class Example {
  private:
    int value;
  public:
    friend void display(Example &e);
  };
void display(Example &e) {
    cout << e.value;
  }
```

4. Provide an example of an inline function.

```
cpp
inline void greet() {
    cout << "Hello, World!";
}
int main() {
```

```
greet();
return 0;
}
```

## 5. Explain the benefits of using friend functions in C++.

 Friend functions provide a way to allow non-member functions to access private and protected data, facilitating certain operations like overloading operators.

## 6. What are the limitations of inline functions?

• Inline functions may increase the size of the binary code, leading to potential performance degradation due to larger code size.

## **Unit-10: Static Members and Polymorphism**

- 1. What are static data members in C++? How are they different from regular data members?
  - Static data members are shared among all instances of a class and are accessed using the class name. They retain their value between function calls.

```
cpp
class Example {
public:
    static int count;
};
int Example::count = 0; // Initialize static member
```

## 2. Explain the use of static member functions in C++.

 Static member functions can access only static data members and other static member functions. They are called using the class name.

```
cpp
class Example {
  public:
    static void display() {
        cout << "Static member function";
    }
</pre>
```

```
};
int main() {
    Example::display();
    return 0;
}
```

## 3. Provide an example of a static member function.

```
cpp
class Counter {
public:
    static int count;
    static void increment() {
        count++;
    }
};
int Counter::count = 0;
int main() {
    Counter::increment();
    cout << Counter::count;
    return 0;
}</pre>
```

# 4. What is polymorphism in C++? Explain with an example.

 Polymorphism allows functions or objects to behave differently based on their context. It can be achieved through function overloading, operator overloading, and inheritance.

```
cpp
class Base {
public:
    virtual void display() {
        cout << "Base class";
    }
};</pre>
```

```
class Derived : public Base {
public:
    void display() override {
        cout << "Derived class";
    }
};
int main() {
    Base *b;
    Derived d;
    b = &d;
    b->display(); // Output: Derived class
    return 0;
}
```

- 5. What is the difference between compile-time polymorphism and run-time polymorphism in C++?
  - Compile-time polymorphism (e.g., function overloading, operator overloading) is resolved during compilation. Run-time polymorphism (e.g., virtual functions) is resolved during program execution.

## 6. Explain how virtual functions are used in C++.

• Virtual functions allow derived classes to override base class methods, enabling dynamic binding and run-time polymorphism.

```
cpp
class Base {
public:
    virtual void display() {
        cout << "Base class";
    }
};
class Derived : public Base {
public:
    void display() override {
        cout << "Derived class";
    }
};</pre>
```

```
int main() {
   Base *b = new Derived();
   b->display(); // Output: Derived class
   delete b;
   return 0;
}
```

These answers cover the essential concepts and provide examples for each question in the respective units. If you have any more specific questions or need further explanations, feel free to ask!

40 continue generating ChatGPT

# **Unit-11: Constructors and Destructors**

## 1. What is a constructor in C++?

• A constructor is a special member function of a class that is automatically called when an object of that class is created. It initializes the object.

```
cpp
class Car {
public:
    string model;
    Car() {
        model = "Default Model";
    }
};
```

## 2. What is a parameterized constructor? Provide an example.

• A parameterized constructor is a constructor that takes arguments to initialize an object with specific values.

cpp
class Car {
public:

```
string model;
Car(string m) {
    model = m;
}
};
int main() {
    Car car1("Tesla");
    cout << car1.model; // Output: Tesla
    return 0;
}
```

## 3. Explain the concept of a copy constructor with an example.

• A copy constructor initializes an object using another object of the same class.

```
cpp
class Car {
  public:
    string model;
    Car(const Car &c) {
       model = c.model;
    }
};
int main() {
    Car car1("Tesla");
    Car car2 = car1;
    cout << car2.model; // Output: Tesla
    return 0;
}
```

## 4. What are dynamic constructors in C++?

 Dynamic constructors allocate memory dynamically for object attributes during runtime using new or other dynamic memory allocation functions.

cpp
class Car {
public:

```
string *model;
Car(const string &m) {
    model = new string(m);
}
~Car() {
    delete model;
}
;
```

## 5. Explain the purpose of a destructor in C++.

 A destructor is a special member function that is automatically called when an object goes out of scope or is explicitly deleted. It cleans up resources allocated to the object.

```
cpp
class Car {
public:
    string *model;
    Car(const string &m) {
        model = new string(m);
    }
    ~Car() {
        delete model;
    }
};
```

#### 6. How are multiple constructors defined in a class? Provide an example.

• Multiple constructors are defined using constructor overloading, allowing different ways to initialize an object.

```
cpp
class Car {
public:
    string model;
    Car() {
        model = "Default Model";
    }
    Car(string m) {
```

```
model = m;
}
};
int main() {
    Car car1;
    Car car2("Tesla");
    cout << car1.model; // Output: Default Model
    cout << car2.model; // Output: Tesla
    return 0;
}</pre>
```

## **Unit-12: More on Constructors and Destructors**

#### 1. What are constructors with default arguments? Provide an example.

 Constructors with default arguments allow parameters to have default values if no argument is provided.

```
cpp
class Car {
  public:
    string model;
    Car(string m = "Default Model") {
       model = m;
    }
};
int main() {
    Car car1;
    Car car2("Tesla");
    cout << car1.model; // Output: Default Model
    cout << car2.model; // Output: Tesla
    return 0;
}
```

# 2. Explain dynamic initialization of objects with an example.

• Dynamic initialization uses runtime values to initialize an object.

срр

```
class Car {
public:
    string model;
    Car(string m) {
        model = m;
    }
};
int main() {
    string userInput;
    cin >> userInput;
    Car car(userInput);
    cout << car.model;
    return 0;
}</pre>
```

## 3. What is the purpose of a destructor, and when is it called?

• A destructor cleans up resources used by an object and is called automatically when the object goes out of scope or is explicitly deleted.

## 4. Provide an example of a class with a destructor.

```
срр
class Car {
public:
    string *model;
    Car(string m) {
        model = new string(m);
    }
    ~Car() {
        delete model;
    }
};
int main() {
    Car car1("Tesla");
    cout << *car1.model;</pre>
    return 0;
}
```

#### 5. Explain the difference between a destructor and a constructor.

• A constructor initializes an object, while a destructor cleans up the resources allocated to an object. Constructors can be overloaded; destructors cannot.

#### 6. What are the rules for defining a destructor in C++?

• A destructor must have the same name as the class, prefixed with a tilde (~), must not have any return type, and cannot take any parameters.

## **Unit-13: Inheritance**

#### 1. What is inheritance in C++?

 Inheritance allows a class (derived class) to inherit properties and behaviors from another class (base class).

```
cpp
class Base {
public:
    void display() {
        cout << "Base class";
    }
};
class Derived : public Base {
};
int main() {
    Derived d;
    d.display(); // Output: Base class
    return 0;
}</pre>
```

#### 2. Explain single inheritance with an example.

• Single inheritance involves one derived class inheriting from one base class.

```
cpp
class Base {
public:
    void display() {
```

```
cout << "Base class";
}
};
class Derived : public Base {
};
int main() {
    Derived d;
    d.display(); // Output: Base class
    return 0;
}</pre>
```

## 3. What is multiple inheritance? Provide an example.

• Multiple inheritance allows a class to inherit from more than one base class.

```
срр
class Base1 {
public:
    void display1() {
        cout << "Base1 class";</pre>
    }
};
class Base2 {
public:
    void display2() {
        cout << "Base2 class";</pre>
    }
};
class Derived : public Base1, public Base2 {
};
int main() {
    Derived d;
    d.display1();
    d.display2();
    return 0;
}
```

#### 4. Explain the concept of multilevel inheritance with an example.

• Multilevel inheritance involves a class derived from another derived class.

```
срр
class Base {
public:
    void display() {
        cout << "Base class";</pre>
    }
};
class Derived1 : public Base {
};
class Derived2 : public Derived1 {
};
int main() {
    Derived2 d;
    d.display(); // Output: Base class
    return 0;
}
```

#### 5. What is hierarchical inheritance? Provide an example.

• Hierarchical inheritance involves multiple derived classes inheriting from a single base class.

```
cpp
class Base {
public:
    void display() {
        cout << "Base class";
    }
};
class Derived1 : public Base {
};</pre>
```

```
class Derived2 : public Base {
};
int main() {
    Derived1 d1;
    Derived2 d2;
    d1.display(); // Output: Base class
    d2.display(); // Output: Base class
    return 0;
}
```

- 6. Explain the concept of hybrid inheritance with an example.
  - Hybrid inheritance is a combination of two or more types of inheritance.

```
срр
class Base1 {
public:
    void display1() {
        cout << "Base1 class";</pre>
    }
};
class Base2 {
public:
    void display2() {
        cout << "Base2 class";</pre>
    }
};
class Derived1 : public Base1, public Base2 {
};
class Derived2 : public Derived1 {
};
int main() {
    Derived2 d;
    d.display1();
    d.display2();
    return 0;
```

## **Unit-14: File Handling**

- 1. What are the different file handling operations in C++?
  - File handling operations include opening, closing, reading, writing, and appending to files.
- 2. How do you open and close a file in C++? Provide an example.

```
cpp
#include <fstream>
using namespace std;
int main() {
    ofstream outFile("example.txt");
    outFile << "Writing to file.";
    outFile.close();
    ifstream inFile("example.txt");
    string content;
    inFile >> content;
    inFile.close();
    return 0;
}
```

#### 3. Explain the different file opening modes in C++.

- o ios::in: Open for reading.
- o ios::out: Open for writing.
- o ios::app: Open for appending.
- o ios::binary: Open in binary mode.
- o ios::ate: Move to the end of the file when opening.
- o ios::trunc: Truncate the file if it exists.

#### 4. What is the difference between text and binary files in C++?

 Text files store data as plain text and can be read by humans. Binary files store data in binary format and are not human-readable. Binary files are generally more efficient for storing complex data.

}

5. Provide an example of reading and writing to a text file in C++.

```
cpp
#include <fstream>
using namespace std;
int main() {
    ofstream outFile("example.txt");
    outFile << "Hello, World!";
    outFile.close();
    ifstream inFile("example.txt");
    string content;
    getline(inFile, content);
    cout << content;
    inFile.close();
    return 0;
}</pre>
```

6. How can you append data to an existing file in C++?

```
cpp
#include <fstream>
using namespace std;
int main() {
    ofstream outFile("example.txt", ios::app);
    outFile << "\nAppending to file.";
    outFile.close();
    return 0;
}</pre>
```