# Aromal S Kunnel

# 22BAI10288

## ASSIGNMENT-III &IV

| | | | |
|---|---|---|---|
| **Branch/Semester** | **B.Tech/Fall semester** | **Session** | **2024-2025** |
| **Name of Faculty** | **Dr. Jitendra P S Mathur** | **Subject** | **Object Oriented Programming With C++** |
| **Module** | **3 and 4** | **Sub Code** | **CSE-2001** |
| **Last date of Submission** | **05.08.2024** | | **Through Google classroom** |

| S.No | Questions | CO Attainment |
|---|---|---|
| 1 | What does inheritance means in c++? What are different forms of inheritance? Give an example of each. | CO.3 |
| 2 | Define polymorphism and Explain Virtual functions with example. What is the difference between static & dynamic binding? | CO.3 |
| 3 | Write C++ program to find sum of three integer float number using template class. | CO.4 |
| 4 | is exception handling? Explain types of exception handling and explain suitable example. | CO.4 |
| 5 | Write a generic function that will sort an array of integer, float value. Create a menu with appropriate options and accept the values from the user. | CO.4 |
| 6 | Write a C++ program involving overriding and function overloading of member function. | CO.4 |
| 7 | Write a C++ program involving multiple catch statements for a try block. | CO.4 |
| 8 | Differentiate between class templates and function templates. | CO.4 |

**A1. Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes (derived classes) based on existing classes (base classes). The derived class inherits the properties and methods of the base class, and can also add its own unique properties and methods. This promotes code reusability, modularity, and hierarchical relationships between classes.**

**There are primarily four types of inheritance in C++:**

**1. Single Inheritance -**

**A derived class inherits from only one base class.**

**Example:**

```
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};
```

**2. Multiple Inheritance -**

**A derived class inherits from more than one base class.**

**Example:**

```
class Mammal {
public:
    void feedMilk() {
        cout << "Mammal is feeding milk" << endl;
```

```
    }
};

class Carnivore {
public:
    void eatMeat() {
        cout << "Carnivore is eating meat" << endl;
    }
};

class Lion : public Mammal, public Carnivore {
public:
    void roar() {
        cout << "Lion is roaring" << endl;
    }
};
```

### 3. Multilevel Inheritance

**A derived class inherits from a base class, and another derived class inherits from the first derived class.**

### Example:

```
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

class Mammal : public Animal {
public:
    void feedMilk() {
        cout << "Mammal is feeding milk" << endl;
    }
};

class Dog : public Mammal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};
```

**4. Hierarchical Inheritance**
**Multiple derived classes inherit from a single base class.**

**Example:**

```
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};

class Cat : public Animal {
public:
    void meow() {
        cout << "Cat is meowing" << endl;
    }
};
```

**A2. Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different types to be treated as if they were of the same type. In simpler terms, it means "many forms." There are two main types of polymorphism in C++:**

**1. Compile-time Polymorphism (Static Binding)**
**This type of polymorphism is resolved at compile time. It's achieved through:**

- **Function Overloading: Multiple functions with the same name but different parameters.**
- **Operator Overloading: Redefining the behavior of operators for user-defined types.**

**Example :**

```cpp
#include <iostream>

using namespace std;

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int main() {
    cout << add(2, 3) << endl; // Calls the int add function
    cout << add(2.5, 3.7) << endl; // Calls the double add function
    return 0;
}
```

**2. Runtime Polymorphism (Dynamic Binding)**
**This type of polymorphism is determined at runtime based on the actual type of the object. It's achieved through:**

- **Virtual Functions: Member functions declared with the virtual keyword in the base class.**

**Example:**

```cpp
#include <iostream>

using namespace std;

class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }
};

class Circle : public Shape {
public:
```

```
    void draw() override {
        cout << "Drawing a circle" << endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing a square" << endl;
    }
};

int main() {
    Shape* shapePtr;
    Circle circle;
    Square square;

    shapePtr = &circle;
    shapePtr->draw(); // Calls the Circle::draw() function

    shapePtr = &square;
    shapePtr->draw(); // Calls the Square::draw() function

    return 0;
}
```

**Static Binding vs. Dynamic Binding**

- **Static Binding:** The compiler determines which function to call based on the type of the object at compile time. This is used in function overloading and operator overloading.
- **Dynamic Binding:** The function to be called is determined at runtime based on the actual type of the object. This is used in virtual functions.

**Key differences:**

- **Resolution time:** Static binding happens at compile time, while dynamic binding happens at runtime.
- **Efficiency:** Static binding is generally more efficient as the compiler can optimize the code.

- **Flexibility:** Dynamic binding provides more flexibility as the actual function to be called can change based on the object's type.

**A3.**

```cpp
#include <iostream>

using namespace std;

template <typename T>
T sum(T a, T b, T c) {
    return a + b + c;
}

int main() {
    int int_sum = sum<int>(3, 4, 5);
    float float_sum = sum<float>(1.2, 3.4, 5.6);

    cout << "Sum of integers: " << int_sum << endl;
    cout << "Sum of floats: " << float_sum << endl;

    return 0;
}
```

**A4.**

**Exception handling** is a mechanism in C++ to handle runtime errors or unexpected conditions that disrupt the normal flow of a program. It helps in preventing program crashes and provides a way to gracefully handle these errors.

**Types of Exception Handling**

In C++, exception handling primarily involves three keywords:

1. **try:** This block encloses the code that might throw an exception.
2. **catch:** This block handles the exception if it occurs within the try block.
3. **throw:** This keyword is used to explicitly throw an exception.

**Example:**

```cpp
#include <iostream>
```

```
using namespace std;

int main() {
    int x = 10, y = 0;

    try {
        if (y == 0) {
            throw "Division by zero";
        }
        int result = x / y;
        cout << "Result: " << result << endl;
    } catch (const char* msg) {
        cerr << "An exception occurred: " << msg << endl;
    }

    return 0;
}
```

## A5.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <typename T>
void sortArray(vector<T>& arr) {
    sort(arr.begin(), arr.end());
}

int main() {
    int choice;
    vector<int> intArray;
    vector<float> floatArray;

    cout << "Menu:\n";
    cout << "1. Enter integers\n";
    cout << "2. Enter floats\n";
    cout << "3. Sort integers\n";
    cout << "4. Sort floats\n";
    cout << "5. Exit\n";
```

```
    while (true) {
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: {
                int num;
                cout << "Enter integers (enter -1 to stop): ";
                while (true) {
                    cin >> num;
                    if (num == -1) {
                        break;
                    }
                    intArray.push_back(num);
                }
                break;
            }
            case 2: {
                float num;
                cout << "Enter floats (enter -1 to stop): ";
                while (true) {
                    cin >> num;
                    if (num == -1) {
                        break;
                    }
                    floatArray.push_back(num);
                }
                break;
            }
            case 3:
                sortArray(intArray);
                cout << "Sorted integers: ";
                for (int num : intArray) {
                    cout << num << " ";
                }
                cout << endl;
                break;
            case 4:
                sortArray(floatArray);
                cout << "Sorted floats: ";
                for (float num : floatArray) {
                    cout << num << " ";
                }
                cout << endl;
                break;
```

```
            case 5:
                exit(0);
            default:
                cout << "Invalid choice\n";
        }
    }

    return 0;
}
```

## A6.

```
#include <iostream>

using namespace std;

class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }

    void area(int side) {
        cout << "Area of shape with side " << side << endl;
    }

    void area(int length, int breadth) {
        cout << "Area of shape with length " << length << " and breadth " <<
breadth << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle" << endl;
    }

    void area(int radius) {
        float area = 3.14 * radius * radius;
        cout << "Area of circle: " << area << endl;
    }
};
```

```
int main() {
    Shape* shapePtr;
    Circle circle;

    shapePtr = &circle;

    shapePtr->draw(); // Overridden function called
    shapePtr->area(5); // Ambiguous call, will result in an error

    circle.draw(); // Overridden function called
    circle.area(5); // Calls Circle's area(int)

    return 0;
}
```

## A7.

```
#include <iostream>
#include <stdexcept>

using namespace std;

int main() {
    int numerator, denominator;

    try {
        cout << "Enter numerator: ";
        cin >> numerator;
        cout << "Enter denominator: ";
        cin >> denominator;

        if (denominator == 0) {
            throw runtime_error("Division by zero");
        }

        if (numerator < 0 || denominator < 0) {
            throw invalid_argument("Negative values not allowed");
        }

        int result = numerator / denominator;
        cout << "Result: " << result << endl;
    } catch (const runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
    } catch (const invalid_argument& e) {
```

```
        cerr << "Error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "Unexpected error: " << e.what() << endl;
    } catch (...) {
        cerr << "Unknown error occurred." << endl;
    }

    return 0;
}
```

**A8.**

**Class Templates vs. Function Templates**

**Both class templates and function templates are powerful tools in C++ that allow for generic programming, enabling code reusability across different data types. However, they have distinct purposes and characteristics.**

**Function Templates**

- **Purpose: Create generic functions that can work with different data types.**
- **Syntax:**

**Example:**

```
template <typename T>
T myFunction(T a, T b) {
    // Function body
}
```

- **Characteristics:**
  - **Operates on data, not objects.**
  - **Type deduction is automatic based on function arguments.**
  - **Cannot access private members of class templates.**
  - **Can be overloaded.**

**Class Templates**

- **Purpose: Create generic classes that can be instantiated with different data types.**
- **Syntax:**

**Example:**

```
template <typename T>
class MyClass {
    // Class members
};
```

**Characteristics:**

- **Operates on objects.**
- **Requires explicit type specification when creating objects.**
- **Can access private members of its instances.**
- **Can be specialized for specific data types.**