C++ ASSIGNMENT

Inheritance and Types of Inheritance

SUBMITTED BY: Rahul Kumar (85)

Assignment-Solutions

Section 1: Inheritance and Types of Inheritance

1. Single Inheritance

A. Employee-Manager

```
#include <iostream>
#include <string>

class Employee {
protected:
    std::string name;
    int id;
    double salary;

public:
    Employee(std::string name, int id, double salary) : name(name), id(id), salary(salary) {}

    void displayDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "ID: " << id << std::endl;
        std::cout << "Salary: " << salary << std::endl;
    }

    double getSalary() const {
        return salary;
    }
};

class Manager : public Employee {
private:
    double bonus;
```

```
public:
    Manager(std::string name, int id, double salary, double bonus) : Employee(name, id, salary), bonus(bonus) {}

    double calculateTotalSalary() const {
        return getSalary() + bonus;
    }

    void displayDetails() const {
        Employee::displayDetails();
        std::cout << "Bonus: " << bonus << std::endl;
        std::cout << "Total Salary: " << calculateTotalSalary() << std::endl;
    }
};

int main() {
    Manager manager("Rahul Kumar", 101, 60000.0, 15000.0);
    manager.displayDetails();
    return 0;
}
```

Example Output:

```
Name: Rahul Kumar
ID: 101
Salary: 60000
Bonus: 15000
Total Salary: 75000
```

B. Person-Student-Teacher

```
#include <iostream>
#include <string>

class Person {
protected:
    std::string name;
    int age;

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

    void displayDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
    }
};

class Student : public Person {
private:
    int studentID;
    std::string course;

public:
    Student(std::string name, int age, int studentID, std::string course) : Person(name, age), studentID(studentID), course(course) {}

    void displayDetails() const {
        Person::displayDetails();
        std::cout << "StudentID: " << studentID << std::endl;
        std::cout << "Course: " << course << std::endl;
    }
```

```
};

class Teacher : public Person {
private:
    int teacherID;
    std::string subject;

public:
    Teacher(std::string name, int age, int teacherID, std::string subject) : Person(name, age), teacherID(teacherID), subject(subject) {}

    void displayDetails() const {
        Person::displayDetails();
        std::cout << "TeacherID: " << teacherID << std::endl;
        std::cout << "Subject: " << subject << std::endl;
    }
};

int main() {
    Student student("Alice", 20, 12345, "Computer Science");
    Teacher teacher("Bob", 35, 67890, "Mathematics");

    student.displayDetails();
    std::cout << std::endl;
    teacher.displayDetails();

    return 0;
}
```

Example Output:

Name: Alice

Age: 20

StudentID: 12345

Course: Computer Science

Name: Bob

Age: 35

TeacherID: 67890

Subject: Mathematics

C. BankAccount-SavingsAccount

```
#include <iostream>
#include <string>
class BankAccount {
protected:
  std::string accountNumber;
  double balance;

public:
  BankAccount(std::string accountNumber, double initialBalance) :
accountNumber(accountNumber), balance(initialBalance) {}

  void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
      std::cout << "Deposit successful. New balance: " << balance << std::endl;
    } else {
      std::cout << "Invalid deposit amount." << std::endl;
    }
  }

  void withdraw(double amount) {
```

```
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        std::cout << "Withdrawal successful. New balance: " << balance << std::endl;
    } else {
        std::cout << "Insufficient funds or invalid amount." << std::endl;
    }
  }

  double getBalance() const {
    return balance;
  }

  void displayDetails() const {
    std::cout << "Account Number: " << accountNumber << std::endl;
    std::cout << "Balance: " << balance << std::endl;
  }
};

class SavingsAccount : public BankAccount {
private:
  double interestRate;

public:
  SavingsAccount(std::string accountNumber, double initialBalance, double interestRate)
    : BankAccount(accountNumber, initialBalance), interestRate(interestRate) {}

  double calculateInterest() const {
    return getBalance() * interestRate;
  }

  void displayDetails() const {
```

```
    BankAccount::displayDetails();
    std::cout << "Interest Rate: " << interestRate << std::endl;
    std::cout << "Calculated Interest: " << calculateInterest() << std::endl;
  }
};

int main() {
  SavingsAccount savings("123456789", 1000.0, 0.05);
  savings.displayDetails();
  savings.deposit(500.0);
  savings.withdraw(200.0);
  std::cout << "Final Balance: " << savings.getBalance() << std::endl;
  return 0;
}
```

Example Output:

```
Account Number: 123456789
Balance: 1000
Interest Rate: 0.05
Calculated Interest: 50
Deposit successful. New balance: 1500
Withdrawal successful. New balance: 1300
Final Balance: 1300
```

2. Multilevel Inheritance

A. Person-Student-Graduate Student

```
#include <iostream>
#include <string>

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

```
    int age;

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

    void displayDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
    }
};

class Student : public Person {
protected:
    int rollNumber;
    std::string course;

public:
    Student(std::string name, int age, int rollNumber, std::string course) : Person(name, age), rollNumber(rollNumber), course(course) {}

    void displayDetails() const {
        Person::displayDetails();
        std::cout << "Roll Number: " << rollNumber << std::endl;
        std::cout << "Course: " << course << std::endl;
    }
};

class GraduateStudent : public Student {
private:
    std::string thesisTitle;
```

```
public:
    GraduateStudent(std::string name, int age, int rollNumber, std::string course, std::string thesisTitle)
        : Student(name, age, rollNumber, course), thesisTitle(thesisTitle) {}

    void displayDetails() const {
        Student::displayDetails();
        std::cout << "Thesis Title: " << thesisTitle << std::endl;
    }
};

int main() {
    GraduateStudent gradStudent("Charlie", 25, 54321, "Computer Science", "Advanced Algorithms");
    gradStudent.displayDetails();
    return 0;
}
```

Example Output:

```
Name: Charlie
Age: 25
Roll Number: 54321
Course: Computer Science
Thesis Title: Advanced Algorithms
```

B. Animal-Mammal-Dog

```
#include <iostream>
#include <string>

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

```
};

class Mammal : public Animal {
public:
  void walk() {
    std::cout << "Mammal is walking" << std::endl;
  }
};

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

int main() {
  Dog dog;
  dog.eat();
  dog.walk();
  dog.bark();
  return 0;
}
```

Example Output:

```
Animal is eating
Mammal is walking
Dog is barking
```

3. Multiple Inheritance

A. Sports-Academics-Student Performance

```
#include <iostream>
#include <string>

class Sports {
protected:
    std::string sportName;
    int score;

public:
    Sports(std::string sportName, int score) : sportName(sportName), score(score) {}

    void displaySportsDetails() const {
        std::cout << "Sport: " << sportName << std::endl;
        std::cout << "Score: " << score << std::endl;
    }
};

class Academics {
protected:
    std::string subject;
    int marks;

public:
    Academics(std::string subject, int marks) : subject(subject), marks(marks) {}

    void displayAcademicsDetails() const {
        std::cout << "Subject: " << subject << std::endl;
        std::cout << "Marks: " << marks << std::endl;
    }
};
```

```
class StudentPerformance : public Sports, public Academics {
private:
  int totalPerformance;

public:
  StudentPerformance(std::string sportName, int score, std::string subject, int marks)
    : Sports(sportName, score), Academics(subject, marks), totalPerformance(score + marks) {}

  void displayDetails() const {
    displaySportsDetails();
    displayAcademicsDetails();
    std::cout << "Total Performance: " << totalPerformance << std::endl;
  }
};

int main() {
  StudentPerformance student("Football", 90, "Math", 85);
  student.displayDetails();
  return 0;
}
```

Example Output:

```
Sport: Football
Score: 90
Subject: Math
Marks: 85
Total Performance: 175
```

B. Employee-Person-Manager

```
#include <iostream>
#include <string>
```

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

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

    void displayPersonDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
    }
};

class Employee {
protected:
    int employeeID;
    double salary;

public:
    Employee(int employeeID, double salary) : employeeID(employeeID), salary(salary) {}

    void displayEmployeeDetails() const {
        std::cout << "Employee ID: " << employeeID << std::endl;
        std::cout << "Salary: " << salary << std::endl;
    }
};

class Manager : public Person, public Employee {
private:
```

```
    std::string department;

public:
    Manager(std::string name, int age, int employeeID, double salary, std::string department)
        : Person(name, age), Employee(employeeID, salary), department(department) {}

    void displayDetails() const {
        displayPersonDetails();
        displayEmployeeDetails();
        std::cout << "Department: " << department << std::endl;
    }
};

int main() {
    Manager manager("David", 40, 112233, 75000.0, "Sales");
    manager.displayDetails();
    return 0;
}
```

Example Output:

Name: David

Age: 40

Employee ID: 112233

Salary: 75000

Department: Sales

4. Hierarchical Inheritance

```
#include <iostream>
#include <string>

class Vehicle {
```

```
protected:
    std::string brand;
    int year;

public:
    Vehicle(std::string brand, int year) : brand(brand), year(year) {}

    void displayVehicleDetails() const {
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Year: " << year << std::endl;
    }
};

class Car : public Vehicle {
private:
    std::string fuelType;

public:
    Car(std::string brand, int year, std::string fuelType) : Vehicle(brand, year), fuelType(fuelType) {}

    void displayDetails() const {
        displayVehicleDetails();
        std::cout << "Fuel Type: " << fuelType << std::endl;
    }
};

class Bike : public Vehicle {
private:
    int engineCC;

public:
```

```
    Bike(std::string brand, int year, int engineCC) : Vehicle(brand, year), engineCC(engineCC) {}

    void displayDetails() const {
        displayVehicleDetails();
        std::cout << "Engine CC: " << engineCC << std::endl;
    }
};

int main() {
    Car car("Toyota", 2022, "Petrol");
    Bike bike("Honda", 2023, 150);

    car.displayDetails();
    std::cout << std::endl;
    bike.displayDetails();

    return 0;
}
```

Example Output:

Brand: Toyota

Year: 2022

Fuel Type: Petrol

Brand: Honda

Year: 2023

Engine CC: 150

5. Hybrid Inheritance

A. Vehicle-Car/Bike-Sports Car

```
#include <iostream>
#include <string>

class Vehicle {
protected:
    std::string brand;
    int speed;

public:
    Vehicle(std::string brand, int speed) : brand(brand), speed(speed) {}

    void displayVehicleDetails() const {
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Speed: " << speed << std::endl;
    }
};

class Car : virtual public Vehicle {
protected:
    int numDoors;

public:
    Car(std::string brand, int speed, int numDoors) : Vehicle(brand, speed), numDoors(numDoors) {}

    void displayCarDetails() const {
        displayVehicleDetails();
        std::cout << "Number of Doors: " << numDoors << std::endl;
    }
};

class Bike : virtual public Vehicle {
```

```
protected:
    bool hasGear;

public:
    Bike(std::string brand, int speed, bool hasGear) : Vehicle(brand, speed), hasGear(hasGear) {}

    void displayBikeDetails() const {
        displayVehicleDetails();
        std::cout << "Has Gear: " << (hasGear ? "Yes" : "No") << std::endl;
    }
};

class SportsCar : public Car, public Bike {
private:
    bool turbo;

public:
    SportsCar(std::string brand, int speed, int numDoors, bool hasGear, bool turbo)
        : Vehicle(brand,speed), Car(brand, speed, numDoors), Bike(brand, speed, hasGear), turbo(turbo)
{}

    void turboMode() {
        if (turbo) {
            std::cout << "Turbo mode activated!" << std::endl;
        } else {
            std::cout << "Turbo mode not available." << std::endl;
        }
    }

        void displayDetails() const {
        displayVehicleDetails();
```

```
        std::cout << "Number of Doors: " << numDoors << std::endl;
        std::cout << "Has Gear: " << (hasGear ? "Yes" : "No") << std::endl;
        std::cout << "Turbo: " << (turbo ? "Yes" : "No") << std::endl;
    }
};

int main() {
    SportsCar sportsCar("Ferrari", 250, 2, true, true);
    sportsCar.displayDetails();
    sportsCar.turboMode();
    return 0;
}
```

Example Output:

Brand: Ferrari

Speed: 250

Number of Doors: 2

Has Gear: Yes

Turbo: Yes

Turbo mode activated!

B. Person-Student/Teacher-Teaching Assistant

```
#include <iostream>
#include <string>

class Person {
protected:
    std::string name;
    int age;

public:
```

```
    Person(std::string name, int age) : name(name), age(age) {}

    void displayPersonDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
    }
};

class Student : virtual public Person {
protected:
    int studentID;

public:
    Student(std::string name, int age, int studentID) : Person(name, age), studentID(studentID) {}

    void displayStudentDetails() const {
        displayPersonDetails();
        std::cout << "Student ID: " << studentID << std::endl;
    }
};

class Teacher : virtual public Person {
protected:
    std::string subject;

public:
    Teacher(std::string name, int age, std::string subject) : Person(name, age), subject(subject) {}

    void displayTeacherDetails() const {
        displayPersonDetails();
        std::cout << "Subject: " << subject << std::endl;
```

```
    }
};

class TeachingAssistant : public Student, public Teacher {
private:
    std::string labSection;

public:
    TeachingAssistant(std::string name, int age, int studentID, std::string subject, std::string labSection)
        : Person(name, age), Student(name, age, studentID), Teacher(name, age, subject), labSection(labSection) {}

    void displayDetails() const {
        displayStudentDetails();
        displayTeacherDetails();
        std::cout << "Lab Section: " << labSection << std::endl;
    }
};

int main() {
    TeachingAssistant ta("Eve", 28, 98765, "Physics", "Lab A");
    ta.displayDetails();
    return 0;
}
```

Example Output:

```
Name: Eve
Age: 28
Student ID: 98765
Subject: Physics
```

Lab Section: Lab A

Section 2: Dynamic Polymorphism and Virtual Functions

6. Virtual Function for Method Overriding

```
#include <iostream>
class Shape {
public:
   virtual double area() {
      return 0.0;
   }
};

class Circle : public Shape {
private:
   double radius;

public:
   Circle(double radius) : radius(radius) {}

   double area() override {
      return 3.14159 * radius * radius;
   }
};

class Rectangle : public Shape {
private:
   double length;
   double breadth;

public:
   Rectangle(double length, double breadth) : length(length), breadth(breadth) {}
```

```
    double area() override {
        return length * breadth;
    }
};

int main() {
    Shape* shape1 = new Circle(5.0);
    Shape* shape2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << shape1->area() << std::endl;
    std::cout << "Area of Rectangle: " << shape2->area() << std::endl;

    delete shape1;
    delete shape2;

    return 0;
}
```

Example Output:

Area of Circle: 78.5397

Area of Rectangle: 24

7. Pure Virtual Function & Abstract Class

```
#include <iostream>
class Animal {
public:
    virtual void makeSound() = 0; // Pure virtual function
};

class Dog : public Animal {
```

```
public:
    void makeSound() override {
        std::cout << "Dog barks: Woof!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat meows: Meow!" << std::endl;
    }
};

int main() {
    Animal* animal1 = new Dog();
    Animal* animal2 = new Cat();

    animal1->makeSound();
    animal2->makeSound();

    delete animal1;
    delete animal2;

    return 0;
}
```

Example Output:

```
Dog barks: Woof!
Cat meows: Meow!
```

8. Dynamic Method Dispatch Using Virtual Functions

```
#include <iostream>

class BankAccount {
public:
  virtual double calculateInterest() {
    return 0.0;
  }
};

class SavingsAccount : public BankAccount {
private:
  double balance;
  double interestRate;

public:
  SavingsAccount(double balance, double interestRate) : balance(balance),
interestRate(interestRate) {}

  double calculateInterest() override {
    return balance * interestRate;
  }
};

class CurrentAccount : public BankAccount {
public:
  double calculateInterest() override {
    return 0.0;
  }
};

int main() {
```

```
    BankAccount* account1 = new SavingsAccount(1000.0, 0.05);
    BankAccount* account2 = new CurrentAccount();

    std::cout << "Savings Account Interest: " << account1->calculateInterest() << std::endl;
    std::cout << "Current Account Interest: " << account2->calculateInterest() << std::endl;

    delete account1;
    delete account2;

    return 0;
}
```

Example Output:

```
Savings Account Interest: 50
Current Account Interest: 0
```

9. Virtual Destructor

```
#include <iostream>
class Base {
public:
    virtual ~Base() {
        std::cout << "Base class destructor called" << std::endl;
    }
};

class Derived : public Base {
public:
    ~Derived() override {
        std::cout << "Derived class destructor called" << std::endl;
    }
};
```

```
int main() {
    Base* basePtr = new Derived();
    delete basePtr;
    return 0;
}
```

Example Output:

```
Derived class destructor called
Base class destructor called
```

10. Abstract Class with Multiple Derived Classes

```
#include <iostream>
class Employee {
public:
    virtual double calculateSalary() = 0;
};

class FullTimeEmployee : public Employee {
private:
    double monthlySalary;

public:
    FullTimeEmployee(double monthlySalary) : monthlySalary(monthlySalary) {}

    double calculateSalary() override {
        return monthlySalary;
    }
};

class PartTimeEmployee : public Employee {
private:
```

```
    double hourlyWage;
    int hoursWorked;

public:
    PartTimeEmployee(double hourlyWage, int hoursWorked) : hourlyWage(hourlyWage),
hoursWorked(hoursWorked) {}

    double calculateSalary() override {
        return hourlyWage * hoursWorked;
    }
};

int main() {
    Employee* employee1 = new FullTimeEmployee(5000.0);
    Employee* employee2 = new PartTimeEmployee(25.0, 20);

    std::cout << "Full Time Employee Salary: " << employee1->calculateSalary() << std::endl;
    std::cout << "Part Time Employee Salary: " << employee2->calculateSalary() << std::endl;

    delete employee1;
    delete employee2;

    return 0;
}
```

Example Output:

```
Full Time Employee Salary: 5000
Part Time Employee Salary: 500
```

Section 3: Exception Handling

11. Exception Handling: Division by Zero

```
#include <iostream>

double divide(int a, int b) {
   if (b == 0) {
      throw std::runtime_error("Division by zero is not allowed.");
   }
   return static_cast<double>(a) / b;
}

int main() {
   int x, y;
   std::cout << "Enter two integers: ";
   std::cin >> x >> y;

   try {
      double result = divide(x, y);
      std::cout << "Result: " << result << std::endl;
   } catch (const std::runtime_error& error) {
      std::cerr << "Error: " << error.what() << std::endl;
   }

   return 0;
}
```

Example Output:

Enter two integers: 10 0

Error: Division by zero is not allowed.

Enter two integers: 10 2

Result: 5

12. Exception Handling with Multiple Catch Blocks

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

int main() {
  int num;
  std::cout << "Enter an integer: ";
  std::cin >> num;

  try {
    if (num < 0) {
      throw std::invalid_argument("Negative number is not allowed.");
    } else if (num == 0) {
      throw std::logic_error("Zero is not allowed.");
    } else if (num > 1000) {
      throw std::out_of_range("Number is too large ( > 1000).");
    } else {
      std::cout << "Valid number: " << num << std::endl;
    }
  } catch (const std::invalid_argument& error) {
    std::cerr << "Invalid Argument Error: " << error.what() << std::endl;
  } catch (const std::logic_error& error) {
    std::cerr << "Logic Error: " << error.what() << std::endl;
  } catch (const std::out_of_range& error) {
    std::cerr << "Out of Range Error: " << error.what() << std::endl;
  } catch (...) {
    std::cerr << "Unknown exception caught!" << std::endl;
  }
```

```
    return 0;
}
```

Example Output:

Enter an integer: -5

Invalid Argument Error: Negative number is not allowed.

Enter an integer: 0

Logic Error: Zero is not allowed.

Enter an integer: 1001

Out of Range Error: Number is too large (> 1000).

Enter an integer: 500

Valid number: 500

13. Exception Handling in Class Methods

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

class Student {
private:
    std::string name;
    int marks;

public:
    Student(std::string name) : name(name), marks(0) {}

    void setMarks(int m) {
        if (m < 0 || m > 100) {
            throw std::out_of_range("Marks must be between 0 and 100.");
```

```
        }
        marks = m;
    }

    void displayDetails() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Marks: " << marks << std::endl;
    }
};

int main() {
    Student student("Rahul");

    try {
        student.setMarks(105);
        student.displayDetails();
    } catch (const std::out_of_range& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }

    try {
        student.setMarks(85);
        student.displayDetails();
    } catch (const std::out_of_range& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }
    return 0;
}
```

Example Output:

Error: Marks must be between 0 and 100.

Name: Rahul

Marks: 85

1**4. Exception Handling in Constructors**

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

class BankAccount {
private:
  std::string accountNumber;
  double balance;

public:
  BankAccount(std::string accountNumber, double initialBalance) :
accountNumber(accountNumber) {
    if (initialBalance < 0) {
      throw std::invalid_argument("Initial balance cannot be negative.");
    }
    balance = initialBalance;
  }

  void displayDetails() const {
    std::cout << "Account Number: " << accountNumber << std::endl;
    std::cout << "Balance: " << balance << std::endl;
  }
};

int main() {
  try {
    BankAccount account("12345", -100.0);
    account.displayDetails();
```

```
    } catch (const std::invalid_argument& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }

    try {
        BankAccount account("67890", 500.0);
        account.displayDetails();
    } catch (const std::invalid_argument& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }
    return 0;
}
```

Example Output:

```
Error: Initial balance cannot be negative.
Account Number: 67890
Balance: 500
```

15. User-Defined Exception Class

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

class InvalidAgeException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Invalid Age: Age must be 18 or older.";
    }
};

void checkAge(int age) {
```

```
    if (age < 18) {
        throw InvalidAgeException();
    }
    std::cout << "Age is valid." << std::endl;
}

int main() {
    try {
        checkAge(15);
    } catch (const InvalidAgeException& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }

    try {
        checkAge(25);
    } catch (const InvalidAgeException& error) {
        std::cerr << "Error: " << error.what() << std::endl;
    }
    return 0;
}
```

Example Output:

Error: Invalid Age: Age must be 18 or older.

Age is valid.