CL1004 – Object Oriented Programming

Lab

Lab # 8

Inheritance

Instructor: Muhammad Saad Rashad

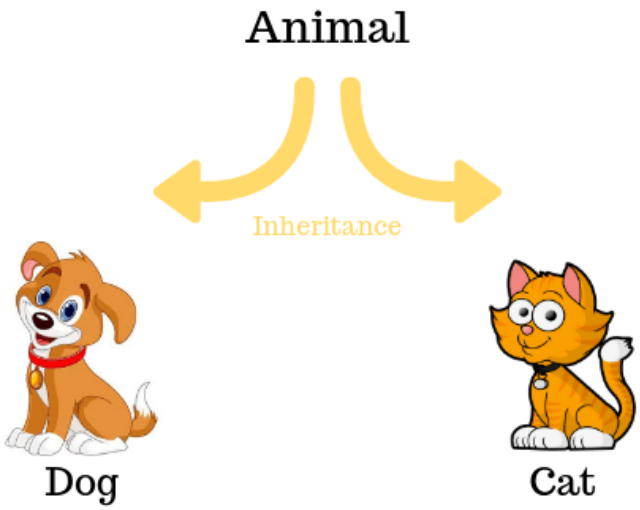
Email: saad.rashad@nu.edu.pk

Department of Computer Science,

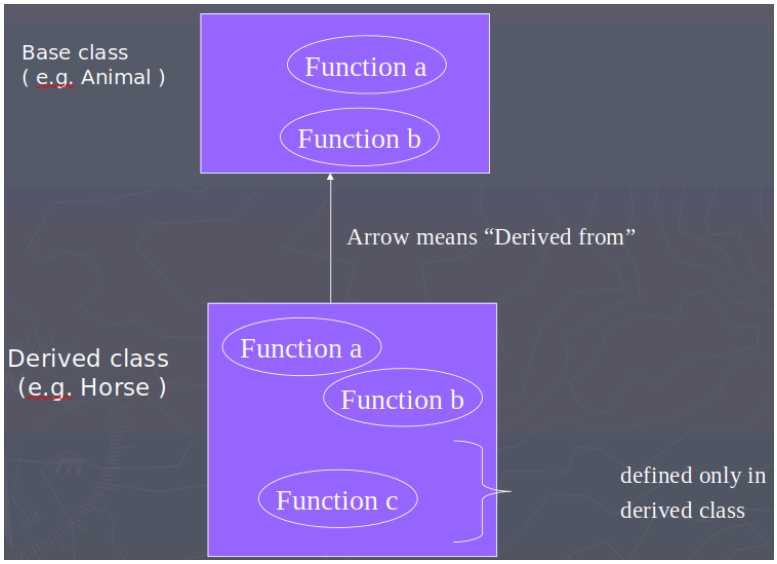
National University of Computer and Emerging Sciences FAST Peshawar

1. **Inheritance:**

- Inheritance is probably the most powerful feature of object oriented programming which is used in achieving software reusability.
- Inheritance is the process of creating new classes, called derived classes from existing or base classes.
- The derived class inherits all the capabilities of the base class but can add its own functionalities and refinements.
- Consider the following figure.
- Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.



-



Example 1:

```
#include<iostream>
using namespace std;
class animal
{
public:
        void eat()
        {
                cout<<"I can eat";
        }
        void sleep()
        {
                cout<<"I can sleep";
        }
};
class dog: public animal
{
        public:
                void bark()
                {
                        cout<<"I can bark";
                }
};
int main()
{
        dog obj;
        obj.eat();
```

```
        cout<<"\n";
        obj.sleep();
        cout<<"\n";
        obj.bark();
}
```

- Base class does not inherit any class.
- Derived class does inherit any other class or classes.
- Single inheritance
    - Derived class only Inherits from one base class
- Multiple inheritance
    - Derived class can Inherit from multiple base classes
    - In this case base classes may be possibly unrelated from a real life point of view.

2. **Protected Access Specifier:**
    - The public members of a class are accessible by all functions in the program and the private members of a class are accessible only by member functions of that class.
    - Similarly, the protected members of a class are accessible by the member functions of that class.
    - The protected members of a base class are, however, accessible by members of its derived classes but the private members of the base class are not accessible directly by members of its derived classes.
    - This is the main difference between the protected and the private access specifiers.
    - The protected members of a base class fall between private and public members.
    - These members are public for the derived class but for the rest of the program, these are treated as private.

Example 2:

```
#include<iostream>
using namespace std;
class animal
{
```

```
protected:
        string specie;
        int age;

public:
        animal()
        {
                age=0;
                specie="unknown";
        }
};
class dog: public animal
{
        private:
                string role;
        public:
                void details(int age_d,string sp_d,string role_r)
                {
                        age = age_d;
                        specie=sp_d;
                        role=role_r;
                }
                void display()
                {
                        cout<<" age: "<<age;
                        cout<<"\n specie: "<<specie;
                        cout<<"\n role: "<<role;
                }
};
int main()
{
        dog obj;
        obj.details(5,"Kangal","Livestock Guardian Dog");
        obj.display();
}
```

3. **Constructors and Destructors in Derived Classes:**

   W.R.T Constructor:

   - When an object of a derived class is instantiated, a **chain of constructor calls** is initiated.

- This means that the **base class constructor is invoked first**, followed by the derived class constructor.
- The **derived class constructor automatically calls the base class constructor**, either **implicitly** (if the base class has a default constructor) or **explicitly** (if the base class has a parameterized constructor and requires arguments).
- In an **inheritance hierarchy**, the base class sits at the top and serves as the foundation for all derived classes. During object creation, the **constructor call starts from the base class and moves downwards to the most derived class**.
- However, during execution, the **last constructor to be called is the first to finish execution** (i.e., the derived class constructor completes execution before the base class constructor).
- Each **base-class constructor initializes its own data members**, which are then **inherited and accessible** by the derived class.
- This ensures that **all inherited attributes are properly initialized before the derived class begins execution**.

W.R.T Destructor:

- When a derived class object is destroyed, a **chain of destructor calls** is initiated in the **reverse order of the constructor call chain**.
- The **destructor of the derived class is called first**, ensuring that any resources specific to the derived class are released.
- Next, the **destructor of the immediate base class** is executed, followed by the destructors of other base classes as we move **up the inheritance hierarchy**.
- This process continues until the **final base class destructor is reached and executed**.
- Once all destructors have completed their execution, the object's memory is fully released, ensuring proper cleanup and preventing memory leaks.

Example 3: Implicit Call Example

```
#include <iostream>
using namespace std;
class Base {
public:
   Base() {
      cout << "Base class constructor called"<<"\n";
```

```
    }
    ~Base() {
        cout << "Base class destructor called"<<"\n";
    }
};
class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor called"<<"\n";
    }
    ~Derived() {
        cout << "Derived class destructor called"<<"\n";
    }
};
int main() {
    cout << "Creating Derived object..." <<"\n";
    Derived obj;
    cout << "End of main function..." <<"\n";
}
```

Example 4: Explicit call

```
#include <iostream>
using namespace std;
class Base {
public:
    Base(int x) {
        cout << "Base class constructor called with value: " << x <<"\n";
    }
    ~Base() {
        cout << "Base class destructor called" <<"\n";
    }
};
class Derived : public Base {
public:
    Derived(int x, int y) : Base(x){  // Explicitly calling Base(x)
        cout << "Derived class constructor called with value: " << y <<"\n";
    }
    ~Derived() {
        cout << "Derived class destructor called" <<"\n";
    }
};
```

```
int main() {
   cout << "Creating Derived object..." <<"\n";
   Derived obj(10, 20);
   cout << "End of main function..." <<"\n";
}
```

4. **Mode of Inheritance W.R.T Access Control:**

   The **specifier (public, protected, or private) controls how the base class members are inherited** in the derived class.

   a. Public inheritance (public)
   *class derived : **public** base*
   - **Public members** of Parent remain **public** in Derived.
   - **Protected members** of Parent remain **protected** in Derived.
   - **Private members** of Parent remain **inaccessible** in Derived.

Example 5:

```
#include <iostream>
using namespace std;

class Parent {
public:
   int x = 10;
protected:
   int y = 20;
private:
   int z = 30;
};

class Derived : public Parent {
public:
   void show() {
      cout<<"x = "<< x<< "\n";  // Accessible (public → public)
      cout<<"y = "<< y<< "\n";  // Accessible (protected → protected)
      // cout << "z = " << z;              Not accessible (private)
   }
};

int main() {
   Derived obj;
```

```
   cout<< obj.x<<"\n";  //Accessible (public)
   // cout << obj.y;     Not accessible (protected)
   obj.show();
}
```

b. Protected inheritance (protected)
Class derived : protected base

- **Public members** of Parent become **protected** in Derived.
- **Protected members** of Parent remain **protected** in Derived.
- **Private members** of Parent remain **inaccessible** in Derived.

Example 6

```
#include <iostream>
using namespace std;

class Parent {
public:
   int x = 10;
protected:
   int y = 20;
private:
   int z = 30;
};

class Derived : protected Parent {
public:
   void show() {
      cout<<"x = "<< x<< "\n";  // Accessible (public → public)
      cout<<"y = "<< y<< "\n";  // Accessible (protected → protected)
      // cout << "z = " << z;              Not accessible (private)
   }
};

int main() {
   Derived obj;
   cout<< obj.x<<"\n";  //Accessible (public)
   // cout << obj.y;     Not accessible (protected)
   obj.show();
}
```

c. private inheritance (private)
Class derived : private base

- **Public members** of Parent become **private** in Derived.
- **Protected members** of Parent become **private** in Derived.
- **Private members** of Parent remain **inaccessible** in Derived.

Example 7: RUN the above program and make the access control private
Class derived : private base

5. **Multilevel VS Multiple Inheritance:**
6. **Function Overriding:**
7. **DMA in base and derived classes:**

Tasks
Q1)

1. Base Class: Person

Design a class named Person, which contains the following data members:

- **Name (string)**
- **Address (string)**

2. Derived Class: Employee *(Inherits from Person)*

Create a class named Employee, which is derived from the Person class. This class should have the following additional attributes:

- **Employee Number (int)**
- **Hours Worked (int)**

Implement constructors, along with appropriate setter and getter functions.

3. Derived Class: ProductionWorker *(Inherits from Employee)*

Create a class named ProductionWorker, which is derived from the Employee class.

This class should include the following additional attributes:

- **Shift (int)**
- **Hourly Pay Rate (double)**
- **Salary (double)**

The Shift variable represents the shift that the employee works:

- **Shift 1 → Day Shift**
- **Shift 2 → Night Shift**

If an employee works more than 5 hours in the night shift, they receive a bonus of 1000.

4. Required Member Functions in ProductionWorker

- **Parameterized Constructors – Initialize object attributes.**
- **Setters and Getters – For all data members.**
- **double calculateSalary() – Computes the employee's salary based on their hourly pay rate and hours worked.**
- **void print_details() – Displays the complete details of the employee.**

5. Implementation in Main Function

- **Create objects of the ProductionWorker class.**
- **Compute the salary using the calculateSalary() function.**
- **Display the employee details using the print_details() function.**

Q2) Scenario: Smart Home Automation

Class Structure and Responsibilities

1. **Device (Base Class)**

This is the parent class for all smart home devices. It contains common attributes and methods that every smart device will inherit.

🔹 **Attributes:**

- **deviceName → Name of the device (e.g., "Living Room Light", "Thermostat")**
- **powerStatus → Whether the device is ON or OFF (bool)**
- **location → Where the device is placed (e.g., "Kitchen", "Bedroom")**

🔹 **Methods:**

- **turnOn() → Turns the device ON**
- **turnOff() → Turns the device OFF**
- **showStatus() → Displays the current status of the device**

---

2. **SmartLight (Derived from Device)**

A smart light that allows users to adjust brightness and change color.

🔹 **Attributes (New for SmartLight):**

- **brightness → Integer value (0-100) representing brightness level**
- **color → String value (e.g., "White", "Blue", "Warm Yellow")**

🔹 **Methods:**

- **adjustBrightness(int level) → Changes brightness level**
- **changeColor(string newColor) → Updates the color of the light**

Example Use Case:
A user wants to set the bedroom light to 50% brightness and change its color to blue.

---

3. **SmartThermostat (Derived from Device)**

A smart thermostat that allows users to adjust temperature and change modes.

🔹 **Attributes (New for SmartThermostat):**

- **temperature → The current temperature setting (e.g., 22°C)**
- **mode → Modes like "Cool", "Heat", or "Auto"**

🔹 **Methods:**

- **setTemperature(float temp) → Updates temperature**
- **switchMode(string newMode) → Changes mode**

Example Use Case:
If the temperature drops below 18°C, the thermostat switches to "Heat" mode automatically.

---

4. **SmartSecurityCamera (Derived from Device)**

A security camera that can record video and provide live feeds.

- 🔹 **Attributes (New for SmartSecurityCamera):**
  - **resolution → Camera resolution (e.g., "1080p", "4K")**
  - **recordingStatus → Whether recording is active (bool)**
- 🔹 **Methods:**
  - **startRecording() → Starts recording**
  - **stopRecording() → Stops recording**
  - **viewFeed() → Displays live video feed**

Example Use Case:
The user turns on the camera at the main entrance and starts recording for security purposes.