Object Oriented Programming (CT-260)
Lab 08

Multiple Inheritance, Diamond Problem, Virtual Inheritance and Virtual Functions

Objectives

The objective of this lab is to familiarize students with the multiple inheritance and the problem related to it. Implementation of Virtual functions associated with inheritance.

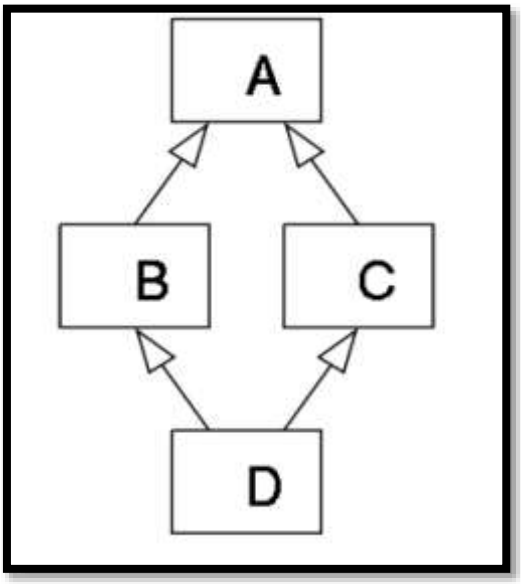
Tools Required

DevC++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Diamond Problem in Hybrid Inheritance:

In case of hybrid inheritance, a Diamond problem may arise. The "dreaded diamond" refers to a class structure in which a particular class appears more than once in a class's inheritance hierarchy.



Example of Diamond Problem:

```
#include <iostream>
using namespace std;
class A{
public:
    int a;
};
class B : public A{
public:
    int b;
};
class C : public A{
public:
    int c;
};
class D : public B, public C{
public:
    int d;
};
int main(){
    D obj;
    obj.a = 200; // will cause an error
}
```

How to Solve the Diamond Problem?

Answer: Virtual Base Classes

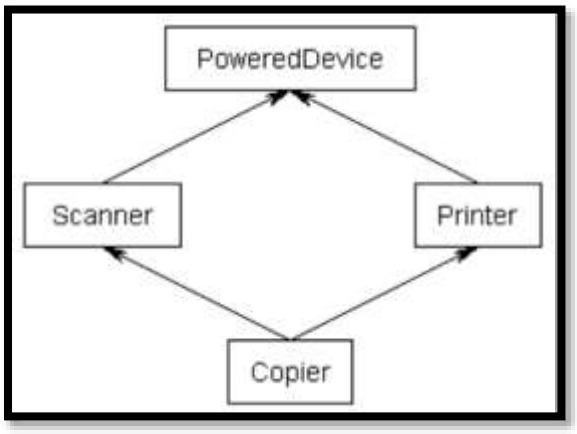
To **share** a base class, simply insert the "virtual" keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only

one base object. The base object is shared between all objects in the inheritance tree and it is only constructed once.

Solving the Diamond Problem:

```
#include <iostream>
using namespace std;
class A{
public:
    int a;
};
class B : virtual public A{ // adding the virtual keyword
public:
    int b;
};
class C : virtual public A{ // adding the virtual keyword
public:
    int c;
};
class D : public B, public C{
public:
    int d;
};
int main(){
    D obj;
    obj.a = 200; // will no longer cause an error
}
```

Diamond Problem with Real Classes and Objects:




```
#include <iostream>
using namespace std;
class PoweredDevice{
public:
    PoweredDevice(int power){
        cout << "PoweredDevice: " << power << '\n';
    }
};
class Scanner : public PoweredDevice{
public:
    Scanner(int scanner, int power) : PoweredDevice(power){
        cout << "Scanner: " << scanner << '\n';
```

```
    }
};
class Printer : public PoweredDevice{
public:
    Printer(int printer, int power) : PoweredDevice(power){
        cout << "Printer: " << printer << '\n';
    }
};
class Copier : public Scanner, public Printer{
public:
    Copier(int scanner, int printer, int power) : Scanner(scanner, power),
            Printer(printer, power) {}
};
int main(){
    Copier copier(1, 2, 3);
    return 0;
}
```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2

--------------------------------
Process exited after 0.2705 seconds with return value 0
Press any key to continue . . .
```

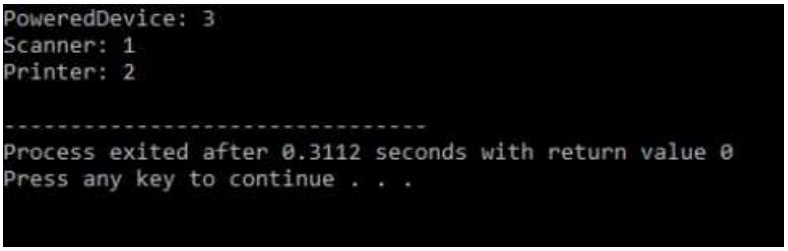
By using Virtual Base Classes:

```
#include <iostream>
using namespace std;
class PoweredDevice
{
public:
    PoweredDevice(int power){
        cout << "PoweredDevice: " << power << '\n';
    }
};
class Scanner : virtual public PoweredDevice{
public:
    Scanner(int scanner, int power) : PoweredDevice(power){
        cout << "Scanner: " << scanner << '\n';
    }
};
class Printer : virtual public PoweredDevice{
public:
    Printer(int printer, int power) : PoweredDevice(power){
        cout << "Printer: " << printer << '\n';
    }
};

class Copier : public Scanner, public Printer{
```

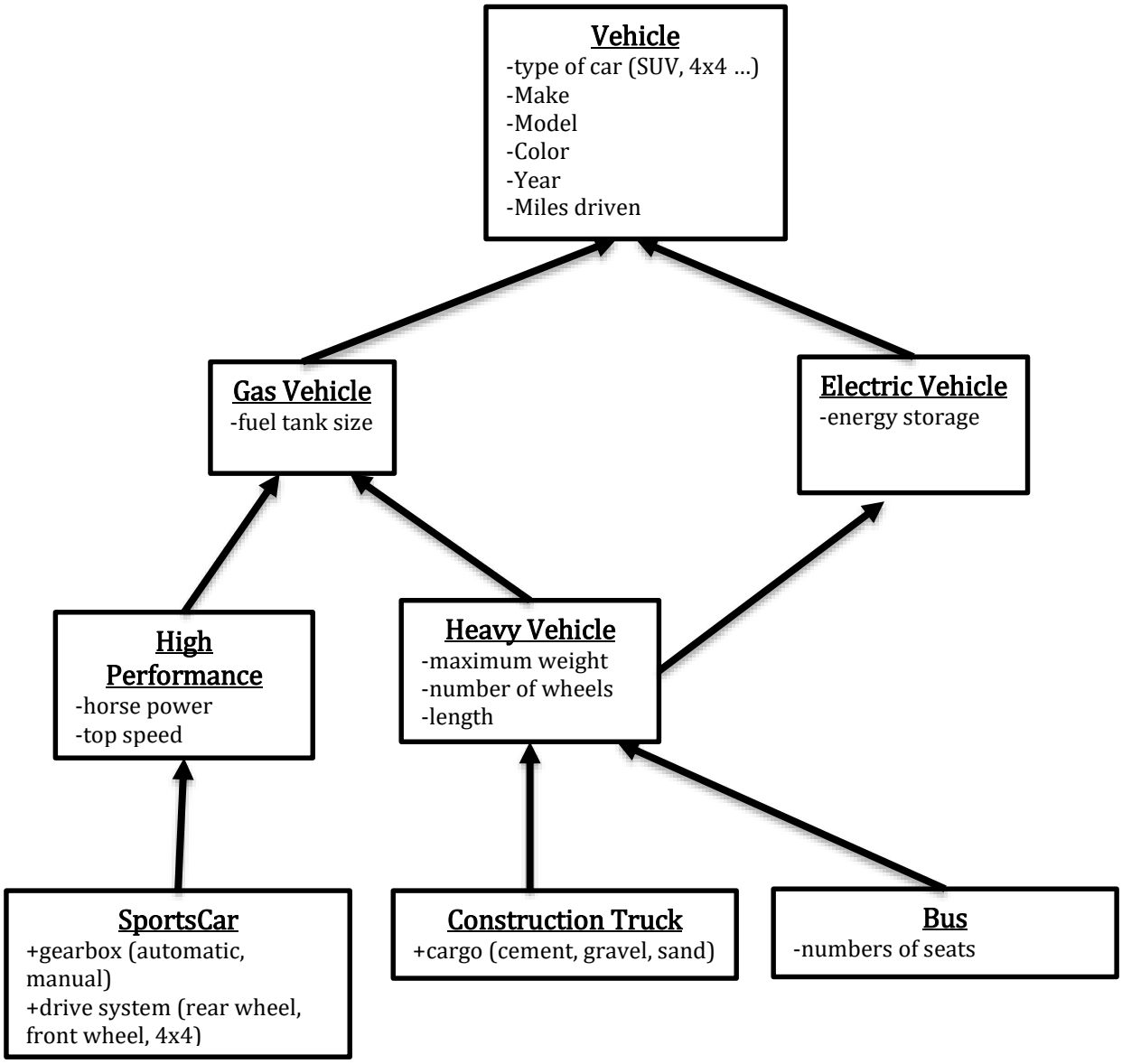
```
public:
    Copier(int scanner, int printer, int power) : PoweredDevice(power),
           Scanner(scanner, power), Printer(printer, power){ }
};
int main(){
    Copier copier(1, 2, 3);
    return 0;
}
```

Now, when you create a Copier class object, you will get only one copy of PoweredDevice per Copier that will be shared by both Scanner and Printer. However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsiblefor creating it?
The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly.



Exercise

1. Implement the given UML class diagram by following the guidelines given below.
   - All the values are required to be set through parameterized constructor
   - Provide necessary accessor methods where required.
   - Create an object of the class bus by initializing it through a parameterized constructor in the main function and display all data members by calling the display function of class bus.



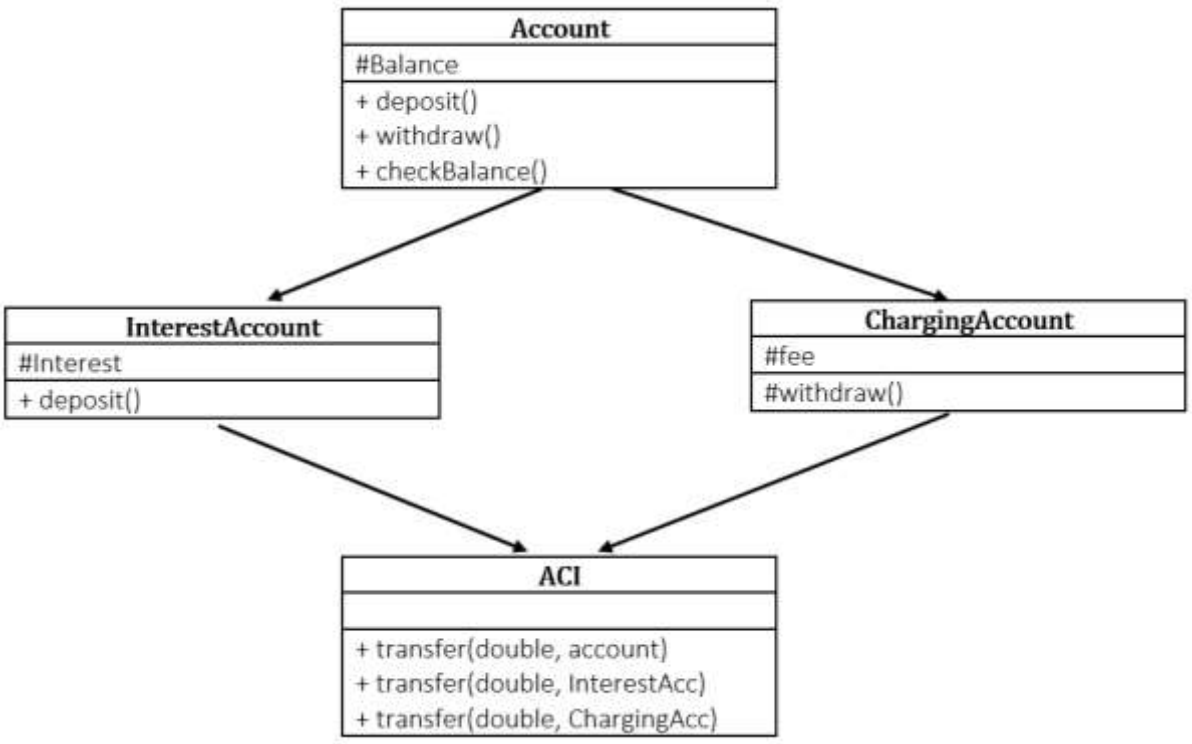
2. Suppose you are designing a game engine for a new video game. The game engine needs to support different types of characters, such as warriors, mages, and archers. Each character type has specific attributes and abilities. The game also includes non-playable characters (NPCs) that have their own unique behaviors. Additionally, there are special characters called "Mighty" that possess both the abilities of a warrior and a mage. To design the character classes and utilize multiple inheritance in this scenario, you can create a class hierarchy with appropriate base classes and derived classes. Here's a possible implementation:
   - Create a base class called Character that contains common attributes and behaviors for all characters, such as name, level, and health.
   - Create derived classes for specific character types: a. Warrior class, derived from Character, which includes attributes and abilities specific to warriors, such as strength, melee

weapons proficiency, and a "slash" ability. b. Mage class, derived from Character, which includes attributes and abilities specific to mages, such as intelligence, spell casting proficiency, and a "fireball" ability. c. Archer class, derived from Character, which includes attributes and abilities specific to archers, such as dexterity, ranged weapons proficiency, and a "rapid shot" ability.

- Create a class called NPC (Non-Playable Character), derived from Character, which includes additional behaviors specific to non-playable characters, such as predefined movement patterns or scripted dialogues.
- Create a class called Mighty, derived from both Warrior and Mage, to represent special Mighty characters. This class inherits attributes and abilities from both Warrior and Mage, allowing the Mighty to possess the combined traits of a warrior and a mage. For example, a Mighty might have high strength and melee prowess like a warrior, as well as the ability to cast powerful spells like a mage.

By using multiple inheritance, you can design a class hierarchy that allows for the sharing of common attributes and behaviors while also enabling specific character types to inherit and extend upon those features. The Mighty class demonstrates the ability to combine attributes and abilities from multiple base classes, showcasing the flexibility of multiple inheritance in this scenario.

3. Implement the given UML class diagram by following the guidelines given below.
   - The interest Account class adds interest for every deposit, assuming a default of 30%.
   - The charging account class charges a default fee of Rs. 25 for every withdrawal.
   - Transfer method of ACI class tasks two parameters: amount to be transferred and object of class in which we have to transfer that amount.
   - Make parameterized constructor and default constructor to take user input for all data members.
   - Make a driver program to test all functionalities.



4. You are developing a software application for a library that manages various types of media, including books, magazines, and DVDs. Each type of media has specific attributes and functionalities, such as the author for books, issue number for magazines, and director for DVDs. Additionally, there are certain operations that are common to all types of media, such as borrowing, returning, and displaying information. Design a class hierarchy using multiple inheritance to handle the different types of media and their functionalities.
   - Create a base class called Media to represent the common attributes and functionalities shared by all types of media. This class will include operations such as borrowing, returning, and displaying information.
   - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.
   - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.
   - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.

   With this class hierarchy, the library application can handle different types of media such as books, magazines, and DVDs. The common functionalities like borrowing, returning, and displaying information will be inherited from the Media class, while the specific attributes and functionalities for each media type will be defined in their respective classes.