NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 103 - COMPUTER PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms.Mahrukh Khan, Ms. Ammara Yaseen, Ms. Tooba Ali, Ms. Maham Mobin, Mr. Muhammad Irfan Ayub

Email: basit.jasani@nu.edu.pk , mahrukh.khan@nu.edu.pk, ammara.yaseen@nu.edu.pk, tooba.ali@nu.edu.pk, maham.mobin@nu.edu.pk, muhammad.irfan@nu.edu.pk

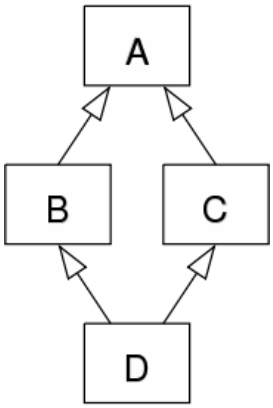
Lab#07

Outline

- Diamond problem in Hybrid Inheritance
- Polymorphism
- Polymorphism Using Function overloading and Function Overriding
- Examples
- Exercise

Diamond Problem

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.



For Example:

```
class Animal
{
public:
voidwalk()
{
cout<<"animal walks";
}

};
```

```
class Donkey : public Animal
{

};
```

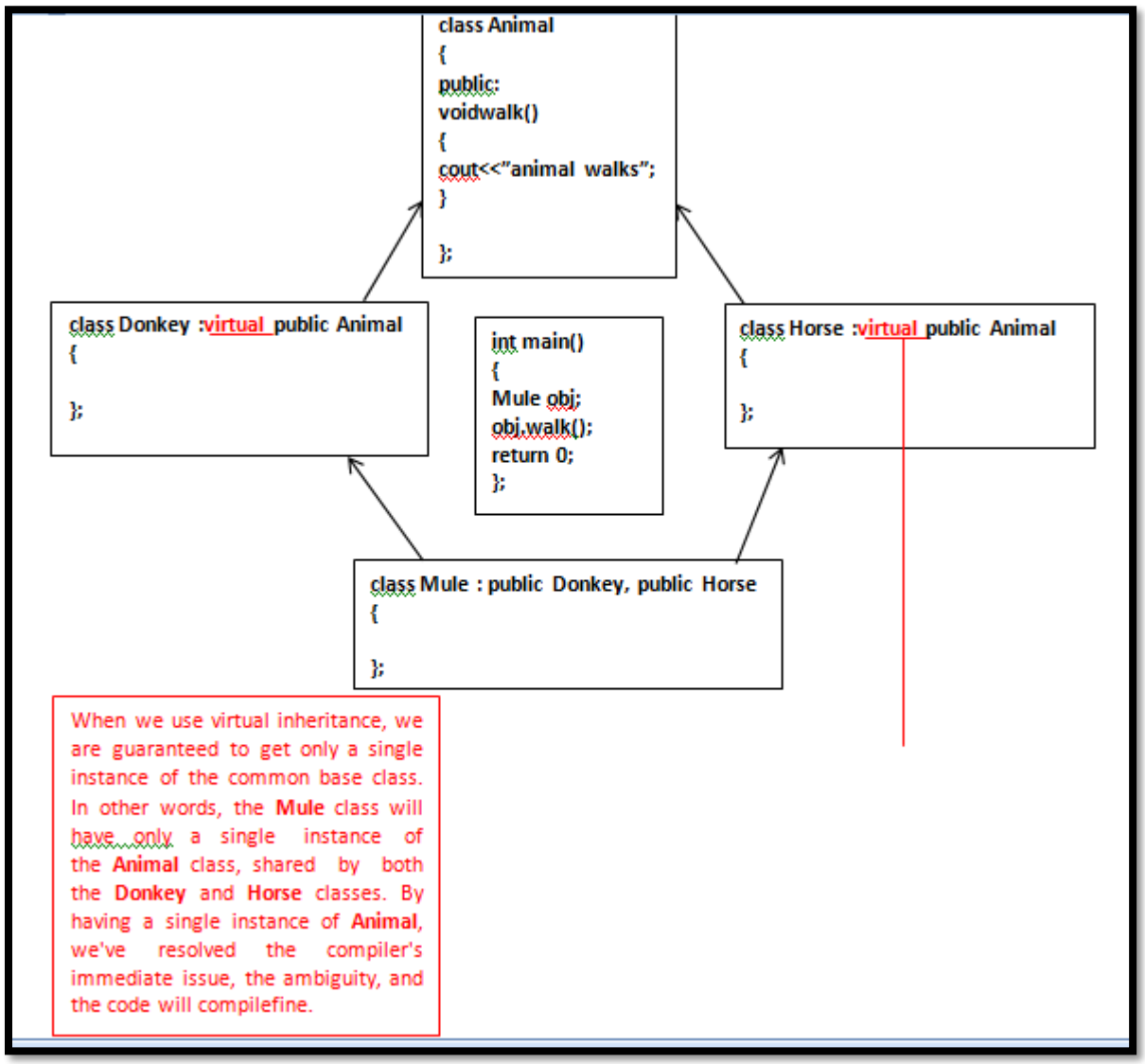
```
int main()
{
Mule obj;
obj.walk();
return 0;
};
```

```
class Horse : public Animal
{

};
```

```
class Mule : public Donkey, public Horse
{

};
```

Call is ambiguous; the compiler can't know which implementation of **walk()** to use, the one from the **Donkey** class or the one from the **Horse** class.

How to solve this Problem??? Virtual Base Class Inheritance



Example#01:

Solution of Diamond Problem Using Virtual Inheritance

```
#include <iostream>

class LivingThing {

public:

   void breathe() {

      std::cout << "I'm breathing as a living thing." <<
std::endl;

   }

};

class Animal : virtual public  LivingThing {

public:

   void breathe() {

      std::cout << "I'm breathing as an animal." << std::endl;

   }

};

class Reptile :virtual public  LivingThing {

public:

void crawl() {

      std::cout << "I'm crawling as a reptile." << std::endl;

   }

};

class Snake :public Animal,public Reptile {

};
```

```
int main() {

Snake snake;

snake.breathe();

   snake.crawl();

 return 0;

 }
```

Example#02:

Parametrized Constructor Calling

```
#include<iostream>

using namespace std;

class Person {

public:

   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }

   Person()    { cout << "Person::Person() called" << endl;   }

};

class Faculty : virtual public Person {

public:

   Faculty(int x):Person(x)   {

      cout<<"Faculty::Faculty(int ) called"<< endl;

   }};

class Student : virtual public Person {

public:

   Student(int x):Person(x) {

      cout<<"Student::Student(int ) called"<< endl;

   }

};

class TA : public Faculty, public Student  {

public:

   TA(int x):Student(x), Faculty(x), Person(x)   {

      cout<<"TA::TA(int ) called"<< endl;

   }

};

int main()  {

   TA t(30);

 }
```

Polymorphism

Polymorphism refers to the ability of a method to be used in different ways, that is, it can take different forms at different times (poly + morphos).

TYPES OF POLYMORPHISM:

There are two types of polymorphism:

- Compile timepolymorphism
- Run timepolymorphism.

COMPILE TIME POLYMORPHISM:

In C++ you can achieve compile time polymorphism by,

- Constructor Overloading (have discussed in the previous labs)
- Function/Method Overloading
- Operator Overloading

Function Overloading

Function Overloading Example

```
#include<iostream>
using namespace std;
class subtraction
{
public:
void difference(int a,int b)
  {
 cout<<a-b<<endl;
}
void difference(int a,int b,int c)
{
cout<<a-b-c<<endl;
}
void difference(double a,double b)
  {
cout<<a-b<<endl;
}

 void difference(int a,double b)
  {
cout<<a-b<<endl;
}

 void difference (double a,int b)
   {
 cout<<a-b<<endl;
}

};
```

```
int main()
{ subtraction obj;
obj.difference(67,34);
obj.difference(4.5,2.3);
obj.difference(12,2,5);
obj.difference(9.4,4);
obj.difference(3,1.4);
return 0;
}
```

Function OverRiding

If we inherit a class into a Derived class and provide definition of base Class function again inside a derived class,then that function said to be overridden and this mechanism is called function overriding.

Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

Requirements For Function Overriding:

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Function Overriding Simple Example

```
#include<iostream>

using namespace std;

class Person {

int id;

public:

   Person(int x)  { id=x;}

   Person()    { cout << "Person::Person() called" << endl;   }

void show()

{cout<<"person's show calling"<<endl;}


};

class Faculty : virtual public Person {int empid;

public:

   Faculty(int x,int y):Person(x)   {

     empid=y;    }};

class Student : virtual public Person {int std_id;

public:

   Student(int x,int y):Person(x) {std_id=y;

         }

void show()

{cout<<"student's show calling"<<endl;}


};
```

```
class Admin : public Faculty, public Student  {

public:

   Admin(int x,int y,int z):Student(x,y), Faculty(x,z), Person(x)   {

         }

void show()

{/*call the Overridden function from overriding function*/

Student::show();

cout<<"admin's show calling"<<endl;}

};

int main()  {

   Admin t(30,40,50);

t.show();

/* call overridden function from the child class*/

t.Person::show();

 }
```

Function Call Binding With Class Objects

Connecting the function call to the function body is called Binding. When it is done before the program is run, its called Early Binding or Static Binding or Compile-time Binding.

Function call Using Objects

```
#include<iostream>
using namespace std;
class Base
{
 public:
 void display()
 {
  cout<<"Base class"<<endl;
 }
};
class Derived:public Base
{
 public:
 void display()
 {
  cout<<"Derived Class"<<endl;
 }
};
```

```
int main()
{
 Base b;      //Base class object
 Derived d;  //Derived class object
 b.display();    //Early Binding Occurs
 d.display();
}
```

Function Call Binding With base Class Pointer

Function call Using Objects

```
#include<iostream>
using namespace std;
class Base
{
 public:
 void display()
 {
  cout<<"Base class"<<endl;
 }
};
class Derived:public Base
{
 public:
 void display()
 {
  cout<<"Derived Class"<<endl;}};
```

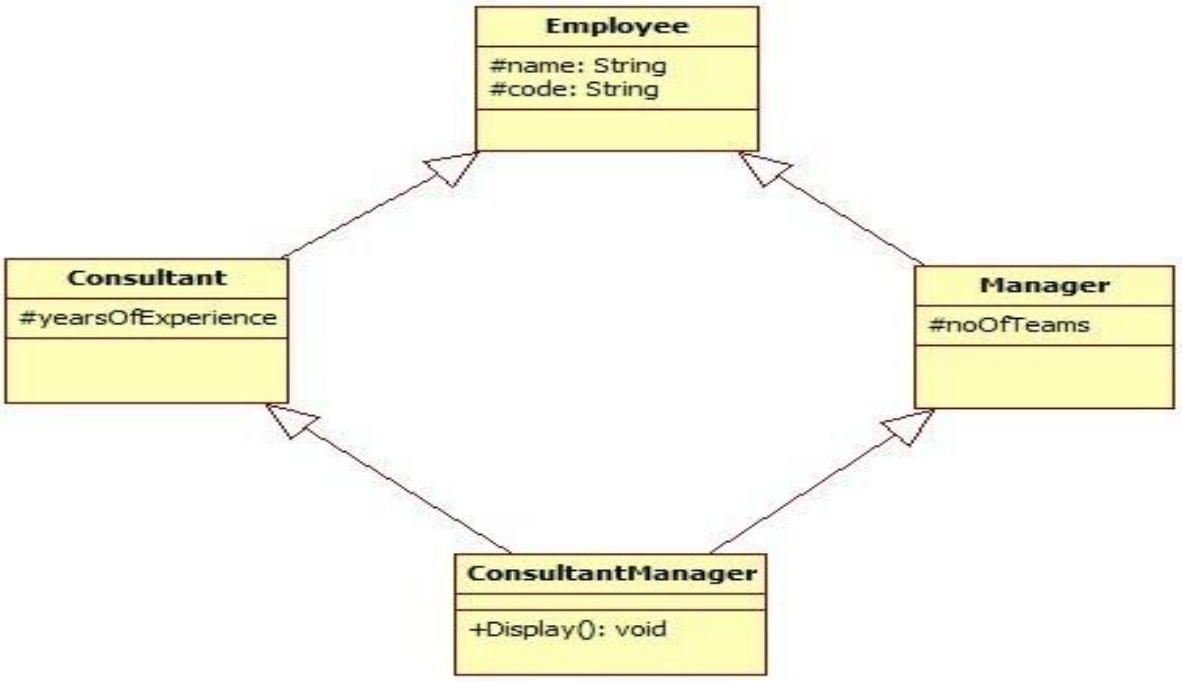
```
   int main()
  {
   Base *b;      //Base class pointer
   Derived d;  //Derived class object
  b=&d;
  b->display();  //early Binding Occurs


}
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.Compiler on seeing **Base class's pointer**, set call to Base class's **display()** function, without knowing the actual object type.

EXERCISES

QUESTION#1

Implement the following scenario in C++:



1. No accessors and mutators are allowed to be used.
2. The Display() function in "ConsultantManager" should be capable of displaying the values of all the data members declared in the scenario (name,code,yearsOfExperience,noOfTeams) without being able to alter thevalues.

3. The "int main()" function should contain only three program statements which are asfollows:

   a) In the first statement, create object of "ConsultantManager" and pass the values for all the data members:

   ConsultantManagerobj("Ali","S-123",17,5);

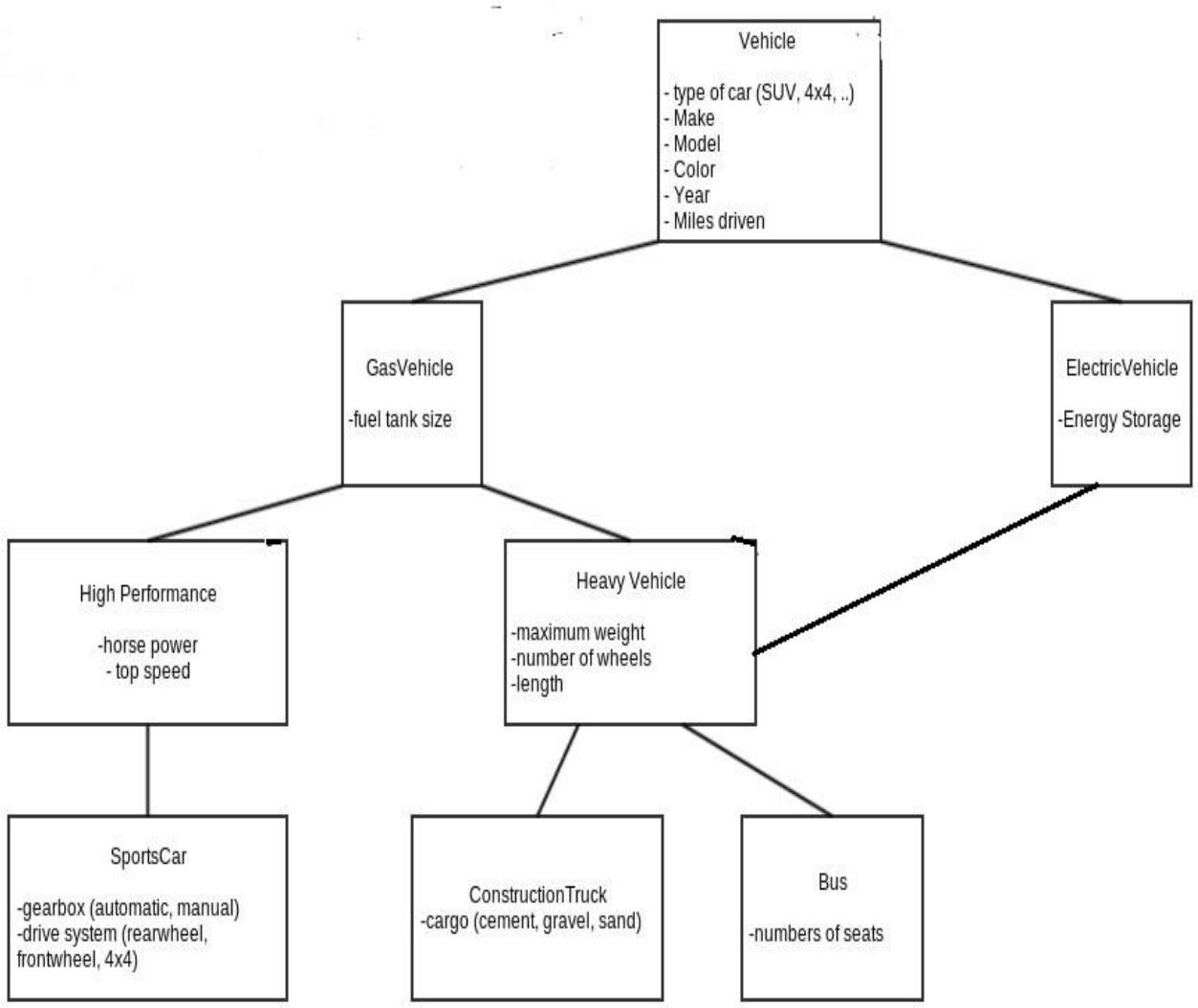
   b) In the second statement, call the Display()function.

   c) In the third statement, return0.

All the values are required to be set through constructors parameter.

QUESTION#2

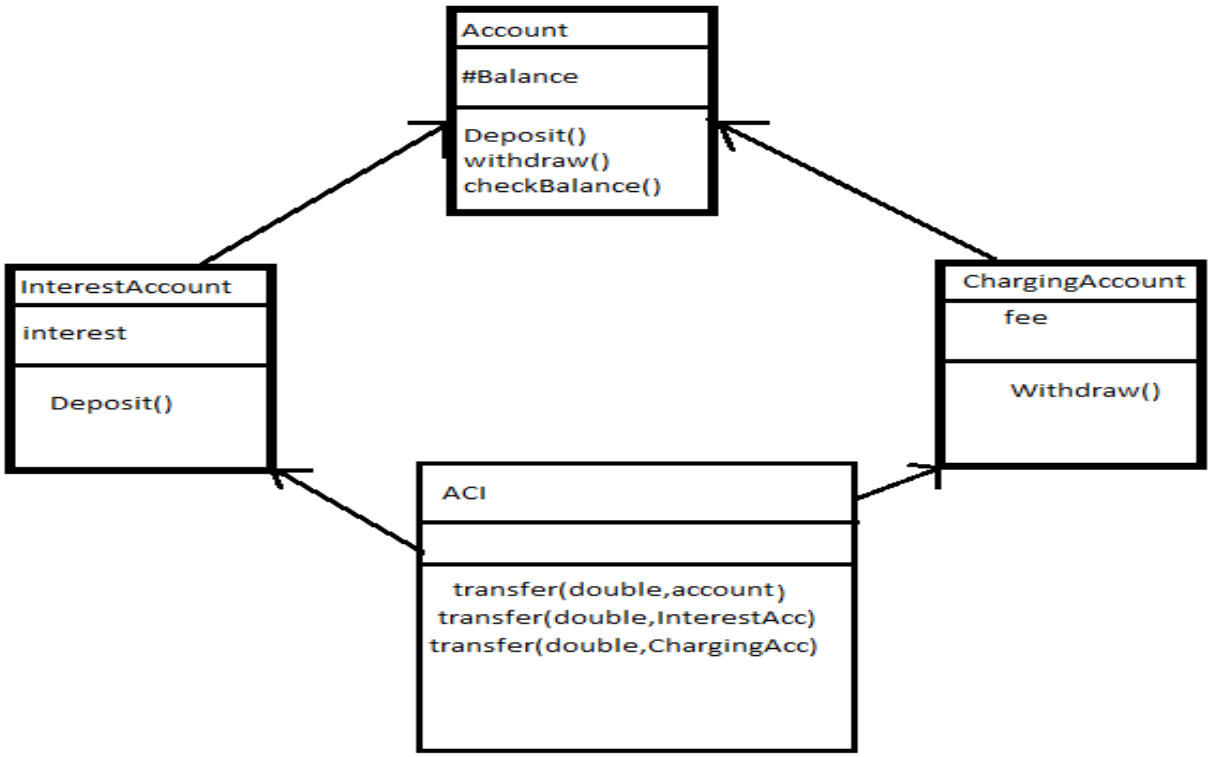
Implement the following scenario in C++:



1. All the values are required to be set through constructors parameter.
2. Provide necessary accessor functions where required.
3. Create an object of class bus by initializing it through parametrized constructor in the main function and display all data members by calling display function of class bus.

QUESTION#3

Implement the following scenario in C++:



1. The interestaccount class adds interest for every deposit,assume a default of 30%.
2. The charging account class charges a default fee of $3 for every withdrawl.
3. Transfer method of aci class takes two parameters,amount to be transfer and object of class in which we have to transfer that amount.
4. Make parametrized constructor, and default constructor to take user input for all data members.
5. Make a driver program to test all functionalities.