**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Fall 2022, Lab Manual - 11**

|  |  |
| --- | --- |
| **Course Code: CS-1004** | **Course: Object Oriented Programming Lab** |
| **Instructor(s) :** |  |

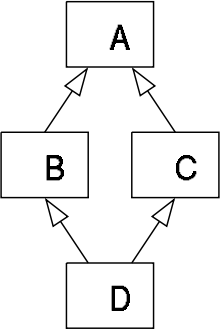
**Lab # 11**

# Outline:

1. Diamond Problem in Hybrid Inheritance
2. Virtual keyword
3. Lab Tasks

## **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;

};

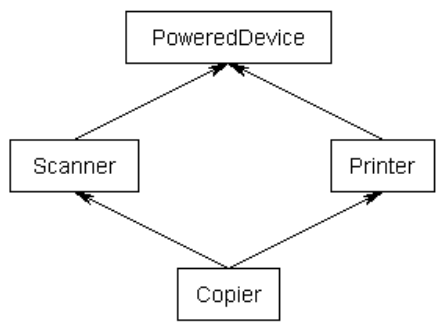
int main() {

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) { }

};

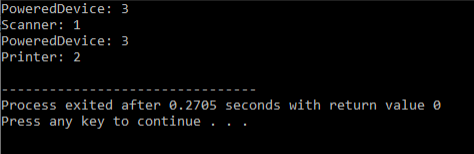
int main() {

Copier copier(1, 2, 3);

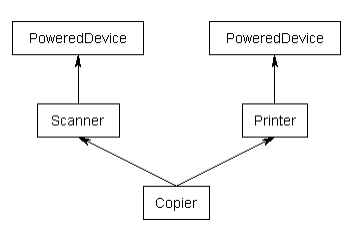
return 0;

}

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:



**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{ // note: PoweredDevice is now a virtual base class

public:

Scanner(int scanner, int power) : PoweredDevice(power){ // this line is required to create Scanner objects, but ignored in this case

cout << "Scanner: " << scanner << '\n';

}

};

class Printer: virtual public PoweredDevice{ // note: PoweredDevice is now a virtual base class

public:

Printer(int printer, int power) : PoweredDevice(power){ // this line is required to create Printer objects, but ignored in this case

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) // PoweredDevice is constructed here at PoweredDevice(power)

{ }

};

int main() {

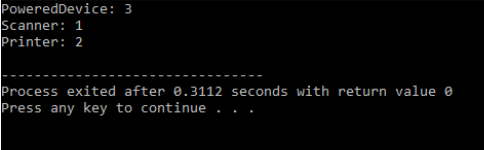
Copier copier(1, 2, 3);

return 0;

}

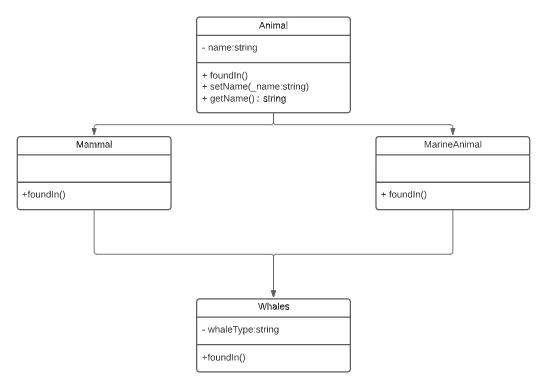
Now, when you create a Copier class object, you will get only one copy of PoweredDevice per Copier that will be shared by both Scanner and Printer. However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it?

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:**

**Task\_01:**

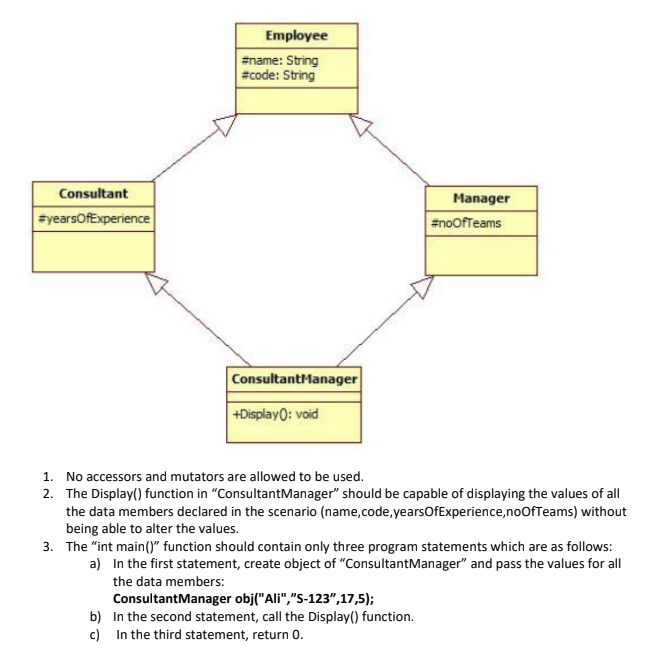
****

Create a class, as depicted in the diagram. Make sure to use virtual base classes for this task. Make an object whaleBaleen. Then set the whaleType to “Baleen”, and name to “whale”

Write the following in the function descriptions for foundIn()

* Animal::foundIn() – Prints: An animal can be found in many places
* Mammal::foundIn() – Prints: A mammal can be found in water or on land
* MarineAnimal::foundIn() – Prints: A marine animal can only be found in bodies of water
* Whales::foundIn() – Prints: A <whaleType> <name> can only be found in the ocean.

**Task\_02:**

****