Inheritance

Today’s Agenda

Today we are going to cover -

- Inheritance basics – base class , dervied class
- Type of inheritance- simple, multi-level, multiple and hierarchical
- Access specifier or mode (private, protected, public inheritance)
- Access specifier (private, protected, public) , Protected members
- Modes (private, protected, public inheritance)
- Overriding member functions,
- Order of execution of constructors and destructors,
- Resolving ambiguities in inheritance,
- Virtual base class.

Let’s Get Started-

Inheritance basics

Inheritance is one of the object oriented programming paradigm as mentioned initially

Inheritance is the process of using properties of one class into the another class

This is achieved by deriving sub-class from the base class.

A class that is inherited is called a super class,base class or parent class and the derived class is called a sub-class, derived class or child class.

A sub-class is a specialized version of a super class.

Eg. we can categories the ‘animal’ into two categories: ‘wild animal’ and ‘pet animal’. Also we can categories ‘wild animal’ into ‘tiger’, ‘lion’, ‘leopard’ and ‘pet animal’ into ‘cat’, ‘dog’, ‘bull’.

Inheritance basics

It inherits all of the instance variables and methods defined by the super class and add its own, unique elements.

Inheritance provides the facility of re-usability.

We can add new features (new data and function) into existing class without modifying it.

This is done by deriving new class (subclass or child class) from existing class (super class or parent class).

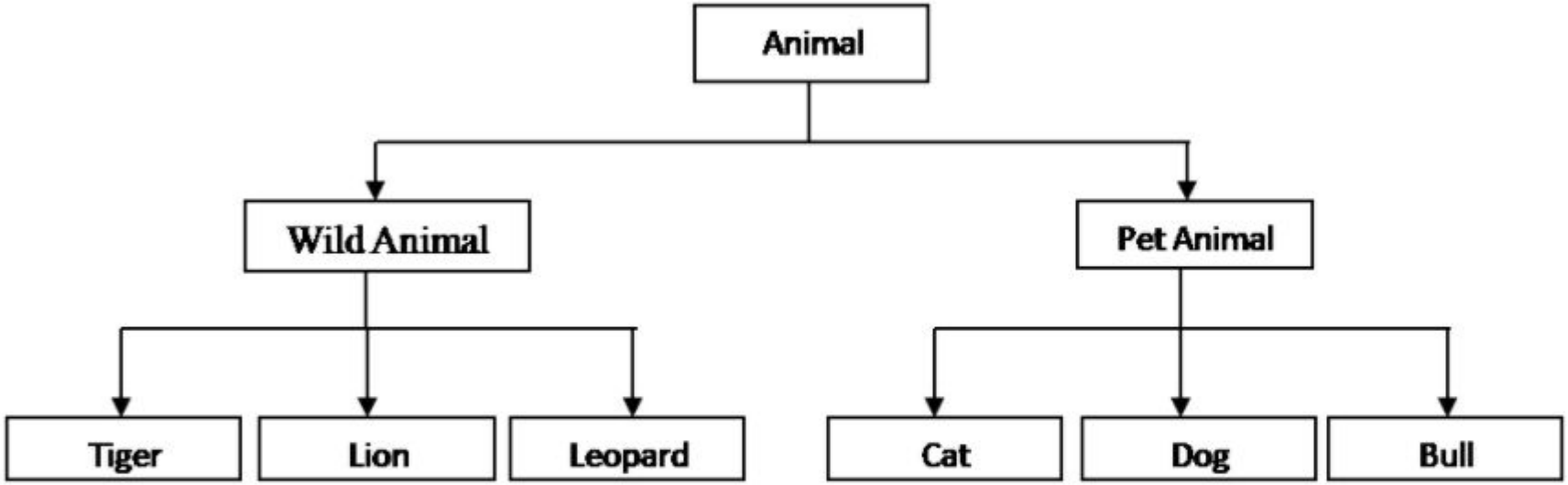
The sub class contains the facility of super class as well as its own features..

Advantages of inheritance

- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Explaination:
Inheritance provides the facility of re-usability. Means Instead of writing the same code, again and again, we can simply inherit the properties of one class into the other. This makes it easier to create and maintain an application. OOP is all about real-world objects and inheritance is a way of representing real-world relationships.

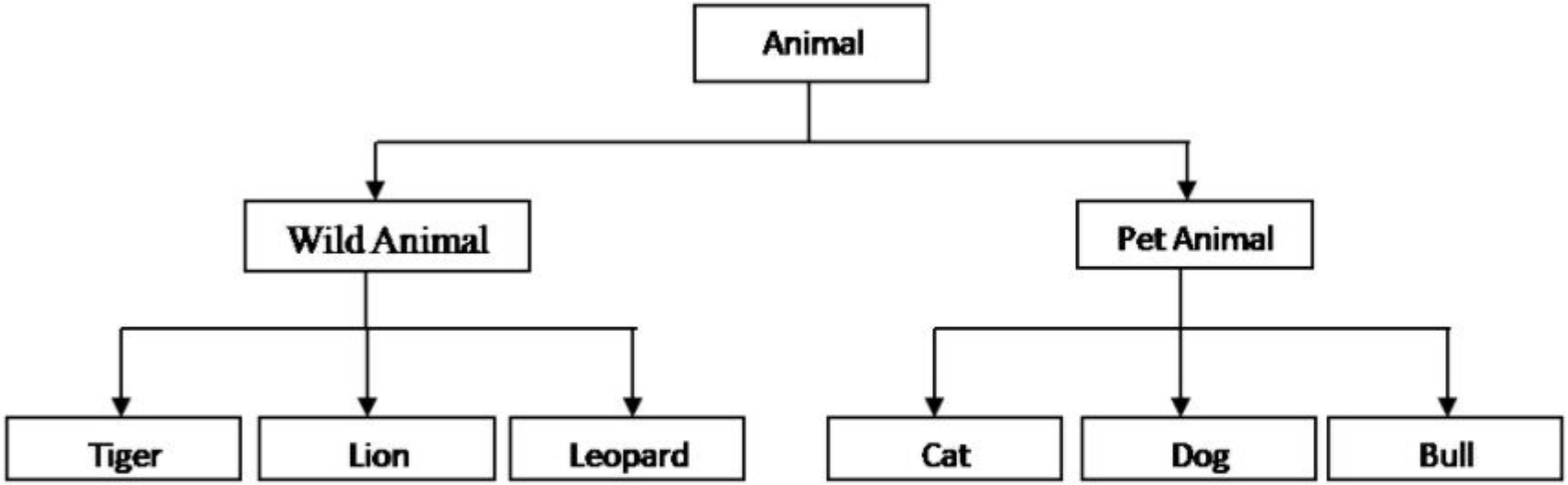
Knowledge check- question



Identify the base class and derived classes in the above figure.
What is ‘Animal’ class called here? What about rest all classes

Type the answers in the chat box.

Knowledge check - Answer



1. Base class : Animal and derived classes :Wild animal and Pet animal;
   Base class : Wild animal , child classes : Tiger, Lion, Leopard;
   Base class : Pet Animal , child classes: Cat, Dog, Bull
2. ‘Animal’ class : Base class /super class / Parent class
   Wild Animal, Pet Animal: Derived class/ Sub class/ Child class.

Types of inheritance

1. Single inheritance : This is a form of inheritance in which a class inherits only one parent class.

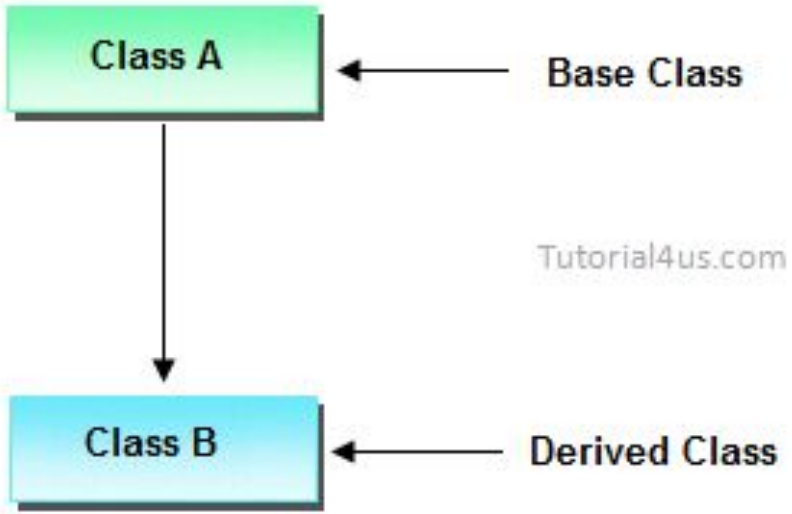
2. Multi-level inheritance : In this form of inheritance , a base class is inherited by a derived class, which further becomes base class and inherited by next level derived class and so on

3. Multiple inheritance : Here a class inherits more than one parent class.

4. Hierarchical inheritance: In this, various child classes inherit a single Parent class.

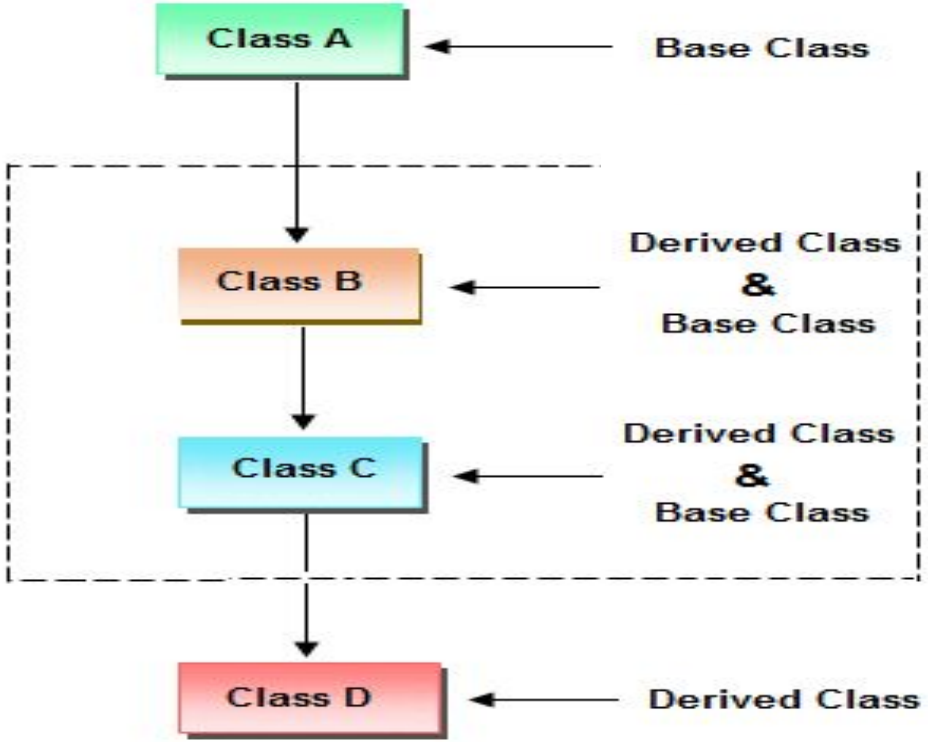
5. Hybrid inheritance: It is the combination of multi-level, multiple and hierarchical inheritance.

Single Inheritance



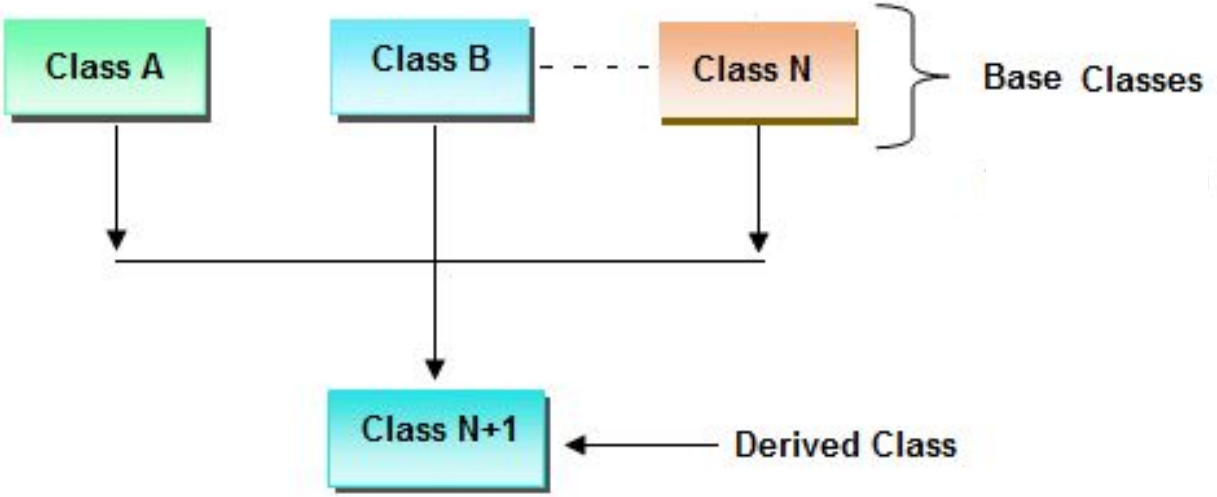
Eg. Parent-child, Animal- Dog, Fruit - Apple , doctor- pediatrician

Multi-level Inheritance



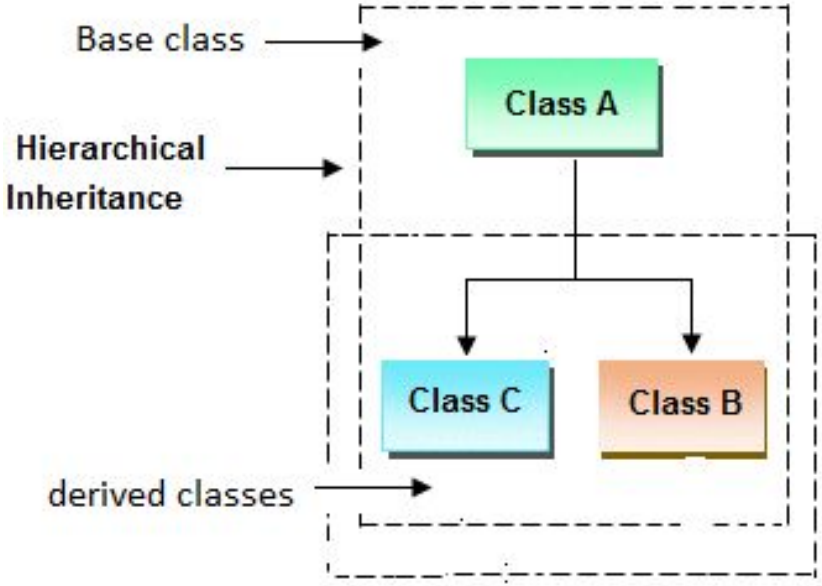
Eg. Grandfather- Father- Child, Vehicle-Car- Audi, Doctor- Orthopedic- KneeSurgeon

Multiple Inheritance



Eg. Mother, Father- Child , student , Teacher- Teaching Assistant

Hierarchical Inheritance

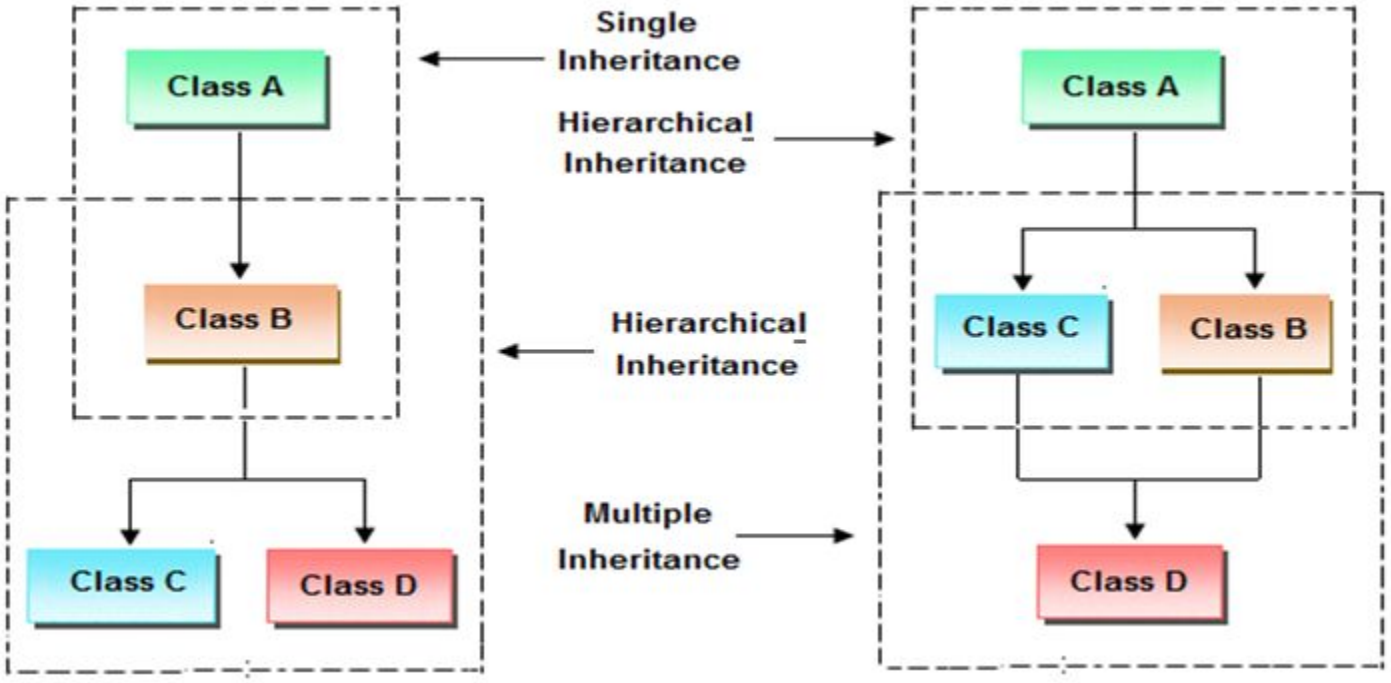


Eg. Animal- Dog, lion, cat etc,

Fruit- Apple, Mango etc,

Person- student ,Teacher, scientist, Engineer etc

Hybrid Inheritance



Eg:Person- student ,Teacher – Teaching Assistant

Syntax of class derivation

```
class BaseClass{
  // members....
  // member function
}

class DerivedClass : public BaseClass{
  // members....
  // member function
}
```

Syntax of class derivation

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes.

To define a derived class, we use a class derivation list to specify the base class(es).

A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class.

If the access-specifier is not used, then it is private by default.

Assignment

Create a class Employee which stores and displays attributes of an employee like empname, empno, department, salary. Create a derived class called Project which allows to store project name. Write a C++ program to create object of project class.

Implementing inheritance

- For creating a sub-class which is inherited from the base class we have to follow the below syntax.

- Syntax:

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

- Here, subclass_name is the name of the sub class.
- access_mode is the mode in which you want to inherit this sub class for example: public, private etc.
- base_class_name is the name of the base class from which you want to inherit the sub class.

Modes of inheritance

- **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

- **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

- **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Let us understand it with example: First understand how private and public members of base class are affected by modes of inheritance

Practice Question - revisited

```
class student
{
        int rollno;
public:
        student() {rollno=1;}
};
class test: public student  //here  public is mode of inheritance
{
        float  marks;
  public:
        test() { marks=40;}
        void display(void);
};
```

Practice Question

```
void test::display()
{
	//cout<<"Rollno ="<<rollno<<endl; //not accessible here as private in base
	cout<<"Marks ="<<marks<<endl;
}
int main()
{
	test t1;
	t1.display();
	return 0;
}
```

Output:

Marks =40

Note that we always create objects of derived class and access all the members of base class and derived class using object of derived class.

Making private members public

```
class student
{
public:
        int rollno;
public:
        student() {rollno=1;}
};
class test: public student
{
public:
        float  marks;
public:
        test() { marks=40;}
        void display(void);
};
```

Making private members public

```
void test::display()
{
	cout<<"Rollno ="<<rollno<<endl; //accessible here as public in base
	cout<<"Marks ="<<marks<<endl;
}
int main()
{
	test t1;
	cout<<"rollno= "<<t1.rollno<<" Marks = "<<t1.marks<<endl;
	t1.display(); //not required now
	return 0;
}
```

Output:

Marks =40

Making private members accessible in derived class

- Solution of making private members public works, but it is against the principle of OOP – Data hiding or encapsulation. Hence you should not make data members of a class public.
- Then what is the solution: how to make base class members accessible in derived class?
- Answer is using by making them protected.
- **Protected members:** The protected members are the members in the base class which can be accessed directly in the derived class. The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Practice Question – protected members

```
class student
{
protected:
        int rollno;
public:
        student() {rollno=1;}
};
class test: public student
{
        float  marks;
  public:
        test() { marks=40;}
        void display(void);
};
```

Practice Question

```
void test::display()
{
	cout<<"Rollno ="<<rollno<<endl; // accessible here as protected in base
	cout<<"Marks ="<<marks<<endl;
}
int main()
{
	test t1;
	t1.display();
	return 0;
}
```

Output:

Rollno=1

Marks =40

How Modes of inheritance impact the members

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Impact on members of modes of inheritance

- **Private members** : Irrespective of mode (type) of inheritance, the private members are not accessible outside the class (not even in main, or further derived classes)

- **Protected members:** If mode of inheritance is public or protected, protected members of base class remain protected in derived class, if mode is private, protected members become private

- **public members:** : If mode of inheritance is public , public members will remain public in derived class . In case of protected mode of inheritance, public members become protected in derived class, if mode is private, public members become private in derived class which cannot be inherited further

Let us understand it with example

Practice Question- observe the inheritance

```
class A
{
public:
        int x;
protected:
        int y;
private:
        int z;
};
class B : public A
{
// x is public
// y is protected
// z is not accessible from B
};
```

Practice Question- observe the inheritance

```
class C : protected A
{
// x is protected
// y is protected
// z is not accessible from C
};

class D : private A    // private' is default for classes
{
 // x is private
 // y is private
 // z is not accessible from D
};
```

MCQ

When the inheritance is private, the private members of the base class are _____ in the derived class

A. Inaccessible
B. Accessible
C. Protected
D. Private

MCQ

When the inheritance is private, the private members of the base class are _____ in the derived class

A. Inaccessible
B. Accessible
C. Protected
D. Private

Answer: option A

Assignment

Create two classes Cuboid and CubiodVol. Cuboid with three data fields-
length, *width* and *height* of *int* types. The class should have d*isplay()* method, to print the length, *width* and *height* of the cuboid separated by space. The*CuboidVol* class is derived from Cuboid class. The class should have *read_input()* method, to read the values of length, *width* and *height* of the Cuboid. The *CuboidVol* class should also the *displayVol()* method to print the volume of the Cuboid (length * width * height).

Output expected:
If length = 12, width = 10 and height = 2
Volume of the cuboid is = (length * width * height)
= 12 * 10 * 2
= 240

Note: Assume necessary data wherever required

Assignment

Use the concept of multi-level inheritance. Create a class student with roll number as a member.
Create 2 classes:
Test: containing the marks of a student in 5 subjects inheriting class student (having roll number of the student).
Result: containing the function Display() to compute the total and average and then displaying the output as Roll number, total and average which are space separated.

Note: Assume necessary data wherever required

Assignment

Create a class shape with attributes as length and breath of float type. Create derived classes rectangle , circle to calculate area of them. Have display methods in both of these derived classes to display the areas calculated .

Note: Assume necessary data wherever required

Overriding member functions

Earlier we have discussed **function overloading** where same function takes various forms.

The function name is same , but the parameter list changes

Now let is see the concept of function overriding

If the member function in defined in both the derived class and the based class with the same name and same number/type of parameters, then the concept is called as **function overriding**

The function in derived class overrides the function in base class.

It is the redefinition of base class function in its derived class with same signature i.e return type and parameters.

Overriding member functions

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

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

Overriding member functions

```
int main() {
Base base1;
base1.print();
return 0;
}
```

Output: Base Function

Had we called the print() function from an object of the Base class, the function would not have been overridden.

Overriding member functions

```
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Output: Derived Function

Here, the same function print() is defined in both Base and Derived classes. So, when we call print() from the Derived object derived1, the print() from Derived is executed by overriding the function in Base. The function was overridden because we called the function from an object of the Derived class.

Access Overriding member functions using ::

Consider above Base and Derived class

```
int main() {
    Derived derived1, derived2;
    derived1.print();

    // access print() function of the Base class
    derived2.Base::print();

    return 0;
}
```

Output:
Derived Function
Base Function

The base class function can be accessed using scope resolution operator.

Order of execution in constructors and destructors

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoked, i.e the order of invokation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

Why the base class's constructor is called first?

Why the base class's constructor is called on creating an object of derived class?

To understand this you will have to recall your knowledge on inheritance.

What happens when a class is inherited from other?

The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only.

So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only.

This is why the constructor of base class is called first to initialize all the inherited members.

Order of constructor call for Multiple Inheritance

For multiple inheritance order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.

```
class student
{
   public:
   student()
   {
      cout << "Inside first base class" << endl;
   }
};
class teacher
{
   public:
    teacher()
   {
      cout << "Inside second base class" << endl;
   }
};
```

Order of constructor call for Multiple Inheritance

```
class TeachingAssistant: public student, public teacher
{
    public:
     // child class's Constructor
    TeachingAssistant()
    {
        cout << "Inside child class" << endl;
    }
};
// main function
int main() {
      // creating object of class Child
     TeachingAssistant TA1;
    return 0;
}
```

Output:

Inside first base class
Inside second base class
Inside child class

Parameterized Constructors in Derived Classes

To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class

The general form of defining a derived class constructor is:

```
Derived-constructor (arglist1, arglist2,…….arglistN):
        base1(arglist1),
        base2(arglist2),
        ……
        ……
        baseN(arglist N)
       {
          Body of derived constructor
        }
```

Parameterized Constructors in Derived Classes

```
#include<iostream>
using namespace std;
class alpha
{
        int x;
public:
alpha(int i)
{
        x=i;
        cout<<"\nalpha initialized\n";
}
void show_x(void)
{
        cout<<"x="<<x<<endl;
}
};
```

Parameterized Constructors in Derived Classes

```
class beta
{
        float y;
        public:
        beta(float j)
        {
                y=j;
                cout<<"beta initialized\n";
        }
        void show_y(void)
        {
                cout<<"y="<<y<<"\n";
        }
};
```

Parameterized Constructors in Derived Classes

```
class gamma:public beta, public alpha
{
        int m, n;
public:
gamma(int a, float b, int c, int d): alpha(a), beta(b)
{
        m=c;
        n=d;
        cout<<"gamma initialized\n";
}
void show_mn(void)
{
        cout<<"m="<<m<<"\n"<<"n="<<n<<"\n";
}
};
```

Parameterized Constructor in Derived Class

```
int main()
{
gamma g(5, 10.75,20,30);
g.show_x();
g.show_y();
g.show_mn();
return 0;
}
```

Output:

```
beta initialized
alpha initialized
gamma initialized
x=5
y=10.75
m=20
n=30
```

Here the constructor is called in the order of inheritance and not in the order of constructor call.

To prove the above point , change the line as follows and observe the output

class gamma:public beta, public alpha

to

class gamma:public alpha, public beta

Points to remember

Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

The parameterised constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterised constructor of sub class.

To call the parameterised constructor of base class inside the parameterised consructor of sub class, we have to mention it explicitly.

The constructor is called in the order of inheritance and not in the order of constructor call

Initialization list in constructors

```
class alpha
{
        int x;
public:
alpha(int i)
{
        x=i;
        cout<<"\nalpha constructed\n";
}
void show_alpha(void)
{
        cout<<"x="<<x<<endl;
}
};
```

Initialization list in constructors

```
class beta
{
        float p,q;
        public:
        beta(float a, float b):p(a), q(b+p)
        {
                cout<<"beta constructed\n";
        }
        void show_beta(void)
        {
                cout<<"p="<<p<<"\n";
                cout<<"q="<<q<<"\n";
        }
};
```

Initialization list in constructors

```
class gamma:public alpha, public beta
{
        int u, v;
public:
gamma(int a, float b, int c): alpha(a*2), beta(c,c), u(a)
{
        v=b;
        cout<<"gamma constructed\n";
}
void show_gamma(void)
{
        cout<<"u="<<u<<"\n"<<"v="<<v<<"\n";
}
};
```

Initialization list in constructors

```
int main()
{
gamma g(2,2.5, 4);
cout<<"Display member values\n";
g.show_alpha();
g.show_beta();
g.show_gamma();
return 0;
}
```

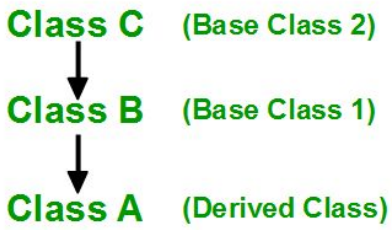
```
Output:
alpha constructed
beta constructed
gamma constructed
Display member values
x=4
p=4
q=8
u=2
v=2
```

Observe how the initializer list works in case of parameterized constructor call in inheritance.

Destructor calls in inheritance

Destructors in C++ are called in the opposite order of that of Constructors.

Order of Inheritance



Order of Constructor Call

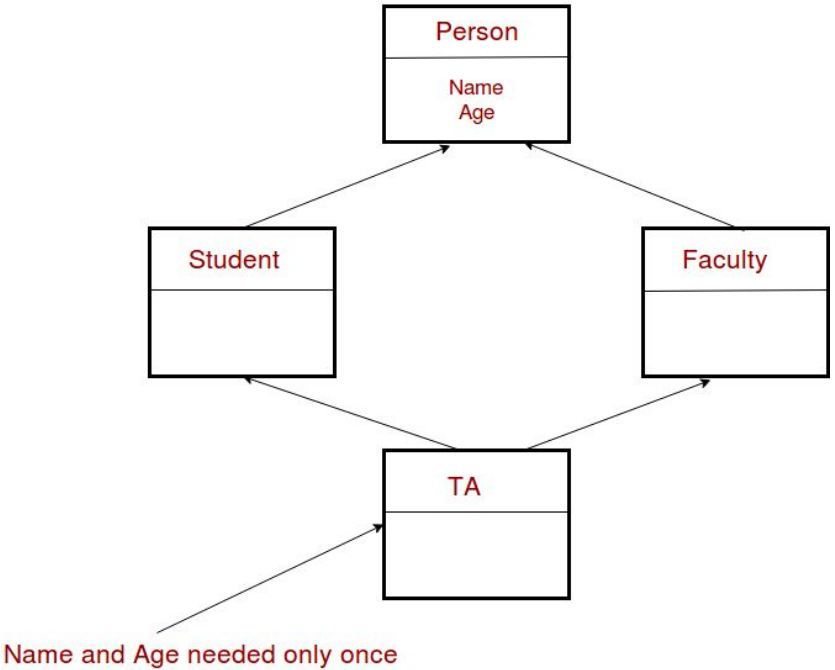
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

Multipath inheritance/diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities. This is a special case of hybrid inheritance



Special case of hybrid inheritance : Multipath inheritance

A derived class with two base classes and these two base classes have one common child class is called multipath inheritance. An ambiguity can arise in this type of inheritance.

```
class ClassA {
public:
    int a;
};
class ClassB : public ClassA {
public:
    int b;
};
class ClassC : public ClassA {
public:
    int c;
};
```

Special case of hybrid inheritance : Multipath inheritance

```
class ClassD : public ClassB, public ClassC {
public:
   int d;
};
void main()
{
   ClassD obj;
    // obj.a = 10;                //Statement 1, Error
    obj.ClassB::a = 10; // Statement 2
   obj.ClassC::a = 100; // Statement 3
    obj.b = 20;
   obj.c = 30;
   obj.d = 40;
    cout << "\n A from ClassB  : " << obj.ClassB::a;
   cout << "\n A from ClassC  : " << obj.ClassC::a;
    cout << "\n B : " << obj.b;
   cout << "\n C : " << obj.c;
   cout << "\n D : " << obj.d;
}
```

Ouput:
A from ClassB : 10
A from ClassC : 100
B : 20
C : 30
D : 40

Special case of hybrid inheritance : Multipath inheritance

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA.

However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, because compiler can't differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

1. Avoiding ambiguity using scope resolution operator:

   Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example. But Still, there are two copies of ClassA in ClassD.

2. Using virtual base class

Virtual Base class

```
class ClassA
{
      public:
      int a;
};
 class ClassB : virtual public ClassA
{
      public:
      int b;
};
class ClassC : public virtual ClassA       //order of public and virtual does not matter
{
      public:
      int c;
};
```

Virtual Base Class

```
class ClassD : public ClassB, public ClassC
{
    public:
    int d;
};
void main()
{
    ClassD obj;
    obj.a = 10;       //Statement 3
    obj.a = 100;      //Statement 4
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<< "\n A : "<< obj.a<<"\n B : "<< obj.b;
    cout<< "\n C : "<< obj.c<< "\n D : "<< obj.d;
 }
```

Output:
A : 100
B : 20
C : 30
D : 40

Note: According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

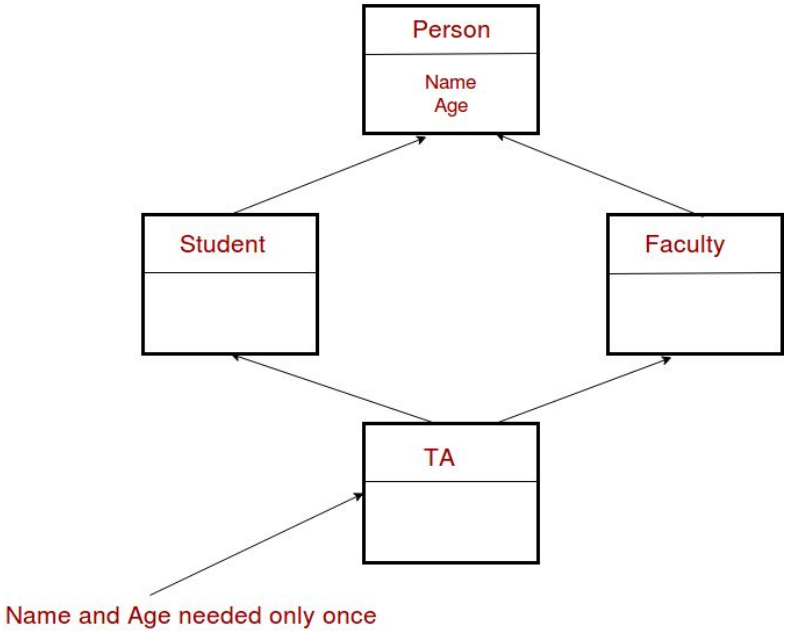
Assignment

Write a c++ program to implement following inheritance

1. without using virtual base class.

Define only constructors at each level of the Inheritance. (need not have any other methods) Observe the order of execution.

2. Using virtual base class.



MCQ

```
#include<iostream>
using namespace std;
 class Base {
public:
   int fun()         {    cout << "Base::fun() called";     }
   int fun(int i)     {   cout << "Base::fun(int i) called";  }
};
 class Derived: public Base  {
public:
   int fun()   {     cout << "Derived::fun() called";   }
};
 int main()  {
   Derived d;
   d.Base::fun(5);
   return 0;
}
```

What is the output:

A. Compiler Error

B. Base::fun(int i) called

MCQ

```
#include<iostream>
using namespace std;
 class Base {
public:
   int fun()        {   cout << "Base::fun() called";    }
   int fun(int i)    {   cout << "Base::fun(int i) called";  }
};
 class Derived: public Base  {
public:
   int fun()   {    cout << "Derived::fun() called";   }
};
 int main()  {
   Derived d;
   d.Base::fun(5);
   return 0;
}
```

What is the output:

A. Compiler Error

B. Base::fun(int i) called

Output: Option B.

We can access base class functions using scope resolution operator.

MCQ

Which one is false?

1. Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

2. The parameterised constructor of base class can be called in default constructor of sub class

3. To call the parameterised constructor of base class, the parameterised consructor of sub class must mention it explicitly.

4. The constructor is called in the order of inheritance and not in the order of constructor call

MCQ

Which one is false?

1. Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

2. The parameterised constructor of base class can be called in default constructor of sub class

3. To call the parameterised constructor of base class, the parameterised consructor of sub class must mention it explicitly.

4. The constructor is called in the order of inheritance and not in the order of constructor call

Answer: Option B

QNA Time

Any Questions??

Thank You!

See you guys in next class.