Practical Lecture : inheritance 3

Quick Recap

Let’s take a quick recap of previous lecture –

- Access specifier (private, protected, public) , Protected members
- Modes (private, protected, public inheritance)
- Overriding member functions,

Today’s

Today we are going to cover -

- Order of execution of constructors and destructors
- Resolving ambiguities in inheritance
- Virtual base class.

Let’s Get Started-

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.

Order of execution in constructors and destructors

```
#include <iostream>
using namespace std;
// base class
class person
{
    public:
    // base class constructor
    person()
    {
        cout << "Inside base class" << endl;
    }
};
// sub class
```

Order of execution in constructors and destructors

```
class student : public person
{
    public:
    //sub class constructor
    student()
    {
        cout << "Inside sub class" << endl;
    }
};
// main function
int main() {
    // creating object of sub class
    student s1;
    return 0;
}
```

Order of execution in constructors and destructors

Ouput:
Inside base class
Inside sub class

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.

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;
}
```

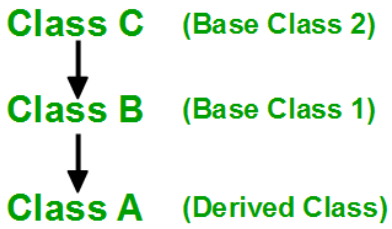
```
Output:
alpha constructed
 beta constructed
   gamma constructed
    Display member values
     x=4
       p=4
        q=8
         u=2
          v=2
```

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

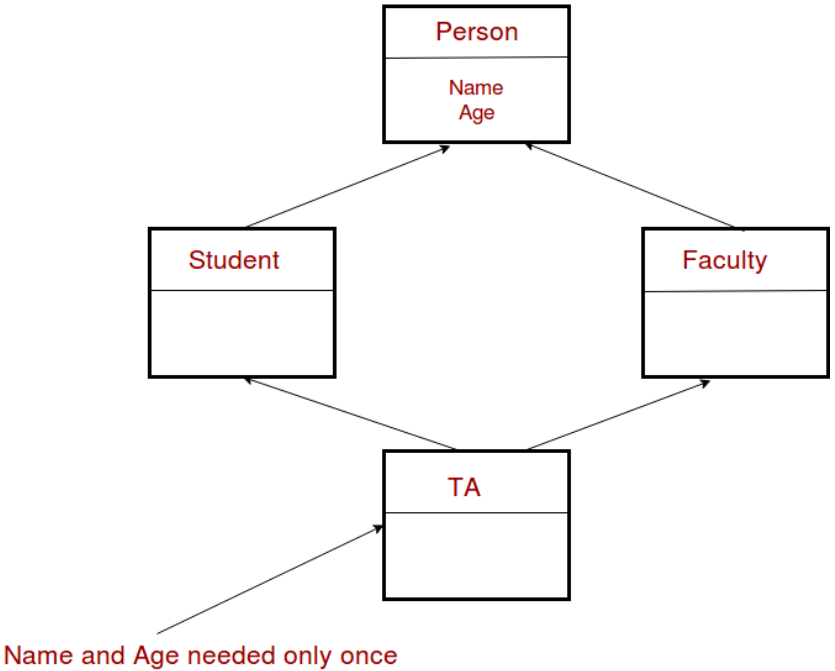
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

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 spe



Special case of hybrid inheritance : Multipath inheritance

A derived class with two base classes and these two base classes have one common base 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

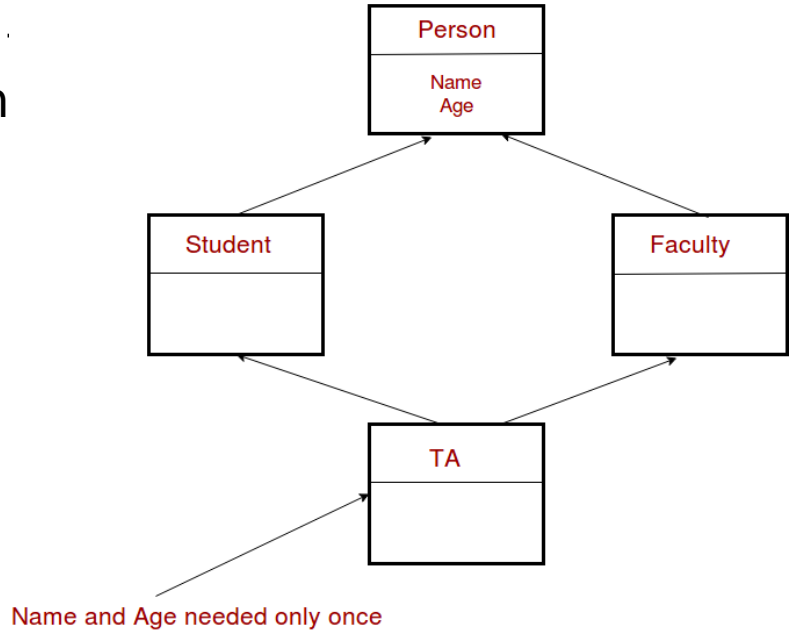
Assignment

Write a c++ program to implement following inheritance

1. without using virtual base class.

Define only constructors at each level of
Inheritance. (need not have any other m
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;
```

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;
```

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


Any
Questions ??

Thank You!

See you guys in next class.