Object Oriented Methodology

Week – *5*, Lecture – *2*

Overriding

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Revisiting virtual functions

A virtual function represents a behaviour of a class, which is “open to extension or modification”

A derived class may be happy with the behaviour it inherited, or, would like to change the same

The changes can be of two types

- A modification means that the derived class no longer requires the definition from the base class …
- … so it replaces it with a new definition
- An extension means that it needs to add some more logic to the definition from the base class …
- … which is often achieved by calling the base class definition from within the new definition

In C++, declaring a method virtual, is a way to flag that this method may have multiple definitions …

- … belonging to different classes in the class hierarchy

This information is useful for the code to exhibit polymorphism (more on this shortly)

Method Overriding

The process of providing a new definition for a virtual method from a base class is called *overriding*

The new definition can either be completely different from the base class definition …

- … or, it may add some logic to the existing base class definition

Method Overriding Scenarios

```
void base::func()
{
    // base class definition
}
```

Overriding

```
void derived::func()
{
    // derived class definition
}
```

Modification of a behaviour

```
void derived::func()
{
    // added logic
    base::func();
    // added logic
}
```

Extension of a behaviour

Usually, you will see these two types of overriding definitions

It is a good idea to first consider the second option – i.e., extending the base class definition by adding logic before and/or after calling it

If you are picking the first option, i.e., providing a fresh definition, be sure that it makes sense to do so !!

Examples for Overriding Scenarios

```
void Toy2::start_playing()
{
        cout<<"You are now playing with "<<name<<" with id "<<id<<endl;
}
```

Definition in the base class

```
void BatteryOperatedToy2::start_playing()
{
        if(is_ready_for_playing())
        {
                cout<<"Switching on the toy..."<<endl;
                in_use = true;
        }
        else
                cout<<"This toy is not yet ready for playing"<<endl;
}
```

Definition in the one derived class

```
void PlaneToy::start_playing()
{
        cout<<"Let's play with a model plane of type "<<model_name<<endl;
        FlyingToy::start_playing();
        BatteryOperatedToy2::start_playing();
        Toy2::start_playing();
}
```

Definition in the another of the derived class

Method Overriding

The process of providing a new definition for a virtual method from a base class is called *overriding*

The new definition can either be completely different from the base class definition …

- … or, it may add some logic to the existing base class definition
- Keep it in mind that the compiler doesn't differentiate between the two scenarios …
- … it is a design choice to pick either of the two options

The virtual keyword plays a subtle, yet an important role in overriding

It is "possible" to redefine a method in a derived class without declaring it as virtual in the base class

- However, this is not considered overriding, but *name hiding*
- Essentially, the definition from the base class is simply "hidden" by the new definition

In some cases, you may be fine with either – overriding or name hiding

- But if you wish to exploit polymorphism features, you need to override a method and not hide it
- The next example displays the subtle difference between the two

Overriding vs Name Hiding

```
class base
{
        public:
                void non_virtual_method();
                virtual void virtual_method();
};
```

```
void base::non_virtual_method()
{
        cout<<"base::non_virtual_method() invoked"<<endl;
}

void base::virtual_method()
{
        cout<<"base::virtual_method() invoked"<<endl;
}
```

Assume that this is our base class (literally !!)

It has two methods – one declared as virtual, the other is non virtual

The definitions contain simple printing statements as shown

Overriding vs Name Hiding

```
class derived : public base
{
        public:
                void non_virtual_method();
                void virtual_method();
};
```

```
void derived::non_virtual_method()
{
        cout<<"derived::non_virtual_method() invoked"<<endl;
}

void derived::virtual_method()
{
        cout<<"derived::virtual_method() invoked"<<endl;
}
```

This is our derived class (again, literally …)

It inherits from the base class, and provides declarations for both the methods from the base

Remember, that for overriding too, you need to have a declaration of the function

Overriding vs Name Hiding

```
base b;
derived d;

base* bp = NULL;
derived* dp = NULL;

cout<<"##########################################"<<endl;
cout<<"Invoking methods directly with objects:"<<endl;
cout<<"----------------------------------------"<<endl;
cout<<"Invoking b.non_virtual_method()"<<endl;
b.non_virtual_method();
cout<<"Invoking b.virtual_method()"<<endl;
b.virtual_method();
cout<<"----------------------------------------"<<endl;
cout<<"Invoking d.non_virtual_method()"<<endl;
d.non_virtual_method();
cout<<"Invoking d.virtual_method()"<<endl;
d.virtual_method();
cout<<"##########################################"<<endl;
```

We create two objects here, one each of both the types

Overriding vs Name Hiding

```
base b;
derived d;

base* bp = NULL;
derived* dp = NULL;

cout<<"#########################################"<<endl;
cout<<"Invoking methods directly with objects:"<<endl;
cout<<"---------------------------------------"<<endl;
cout<<"Invoking b.non_virtual_method()"<<endl;
b.non_virtual_method();
cout<<"Invoking b.virtual_method()"<<endl;
b.virtual_method();
cout<<"---------------------------------------"<<endl;
cout<<"Invoking d.non_virtual_method()"<<endl;
d.non_virtual_method();
cout<<"Invoking d.virtual_method()"<<endl;
d.virtual_method();
cout<<"#########################################"<<endl;
```

We also create two pointers – just to see how polymorphism works here

Overriding vs Name Hiding

```
base b;
derived d;

base* bp = NULL;
derived* dp = NULL;

cout<<"########################################"<<endl;
cout<<"Invoking methods directly with objects:"<<endl;
cout<<"----------------------------------------"<<endl;
cout<<"Invoking b.non_virtual_method()"<<endl;
b.non_virtual_method();
cout<<"Invoking b.virtual_method()"<<endl;
b.virtual_method();
cout<<"----------------------------------------"<<endl;
cout<<"Invoking d.non_virtual_method()"<<endl;
d.non_virtual_method();
cout<<"Invoking d.virtual_method()"<<endl;
d.virtual_method();
cout<<"########################################"<<endl;
```

First, we call these methods using the respective objects

Overriding vs Name Hiding

```
##########################################
Invoking methods directly with objects:
------------------------------------------
Invoking b.non_virtual_method()
base::non_virtual_method() invoked
Invoking b.virtual_method()
base::virtual_method() invoked
------------------------------------------
Invoking d.non_virtual_method()
derived::non_virtual_method() invoked
Invoking d.virtual_method()
derived::virtual_method() invoked
##########################################
```

As shown here, all the method calls, produce expected output

Overriding vs Name Hiding

```
cout<<"#########################################"<<endl;
cout<<"Invoking methods via pointers:"<<endl;
cout<<"-----------------------------------------"<<endl;
bp = &d;
dp = &d;
cout<<"Invoking bp->non_virtual_method()"<<endl;
bp->non_virtual_method();
cout<<"Invoking bp->virtual_method()"<<endl;
bp->virtual_method();
cout<<"Invoking dp->non_virtual_method()"<<endl;
dp->non_virtual_method();
cout<<"Invoking dp->virtual_method()"<<endl;
dp->virtual_method();
cout<<"#########################################"<<endl;
```

Next, we use the two pointers, to point to the derived class object

Remember, it is perfectly fine for `bp` to store a reference to `d`

If we use these pointers (along with pointer to member operator) to invoke the same methods, you can see the difference in the output

Overriding vs Name Hiding

```
########################################
Invoking methods via pointers:
----------------------------------------
Invoking bp->non_virtual_method()
base::non_virtual_method() invoked
Invoking bp->virtual_method()
derived::virtual_method() invoked
Invoking dp->non_virtual_method()
derived::non_virtual_method() invoked
Invoking dp->virtual_method()
derived::virtual_method() invoked
########################################
```

Observe the difference between the output of the class to the virtual and non virtual methods

Despite name hiding, the pointer is able to execute the definition from the base class, even though the object was of the derived type

However, for the virtual method, the pointer checks the actual type of the object “at runtime”, and invoke the correct definition of the method in the inheritance hierarchy

Homework !!

Read more on the difference between overriding and name hiding

- This stackoverflow question has some good answers on this topic: https://stackoverflow.com/questions/19736281/what-are-the-differences-between-overriding-virtual-functions-and-hiding-non-vir