Object Oriented Methodology

Week – *4*, Lecture – *2*

Classes in C++ - Part 3

(Inheriting Virtually, Use of Scope Resolution Operator)

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Multiple Inheritance in C++

C++ supports multiple inheritance, i.e. a class can have more than one base classes

The way to declare multiple base classes is similar to that for a single base class

- You can separate the classes (and the associated access zone) by commas

It is also fine, if two or more base classes have members of the same name …

- … as long as you access them “unambiguously” (more on this shortly)

In addition, there is a provision to protect multiple copies of the same member …

- … for instance, in the case of diamond-shaped inheritance

The constructor calls may become tedious though …

- … especially if the base classes do not have default constructors !!

Yet another type of Toy !!

```
class PlaneToy : public FlyingToy, public BatteryOperatedToy2
{
        private:
                string model_name;

        public:
                PlaneToy(string, int, string, int, string);
                string get_model_name();
                void start_playing();
                void stop_playing();
};
```

This is how we *inherit* multiple classes in the same derived class

The syntax is
`class <D> : <a1> <B>, <a2> <C>, …`
where `D` is the derived class, `B` and `C` are two base classes and a_1 and a_2 are access specifiers

Conceptually, a Plane Toy is a Flying Toy as well as a Battery-Operated Toy

A Plane Toy here represents a battery-operated model of an airplane

Single-level Multiple Inheritance

Let us say there is a class called *Child* which is inheriting from two classes, $Parent_1$ and $Parent_2$

- Also, there may or may not be members in $Parent_1$ and $Parent_2$, which have the same name

It is allowed in C++ to inherit from multiple base classes which may have members with same name

- As long as such members are not accessed in the derived class, the compiler will not complain …
- … in fact, it will create copies of all such members multiple times, one each for each base class

If you want to access these members, you *must* use the scope resolution operator

- For instance, `FlyingToy::start_playing()` and `BatteryOperatedToy2::start_playing()`
- If you try to access such members without the scope resolution operator, you will get a compilation error …
- … where the compiler will tell you that the access is “ambiguous” in nature !!

The constructors may be called in the same fashion, as they are for single inheritance, e.g.

```
PlaneToy::PlaneToy(string s1, int i1, string s2, int i2, string s3)
        : FlyingToy(s1, i1, 2), BatteryOperatedToy2(s1, i1, s2, i2)
```

Virtual Inheritance in C++

Let us say both *Parent1* and *Parent*$_2$, derive from a class called *GrandParent*

- Also assume that there are some inheritable elements in *GrandParent*, which can "travel" to *Child*

Clearly, this is the Diamond Problem – there will be two copies of these elements in *Child* …

- … one via *Parent*$_1$ and another via *Parent*$_2$

You are allowed to do this as well, again, as long as you use the scope resolution operator

However, it may be inefficient (and maybe illogical) to have two copies of *GrandParent* members

C++ allows you to inform about this scenario to the compiler …

- … which can then ensure that a single version of these members reach *Child*

To do so, the information must be provided while creating *Parent*$_1$ and *Parent*$_2$ itself

- By adding the keyword `virtual` before *GrandParent*, you provide the necessary information

Inheriting Virtually

```
class FlyingToy : virtual public Toy2
{
        private:
                bool number_of_wings;

        public:
                FlyingToy(string, int, int);
                void start_playing();
                void stop_playing();
                bool has_wings();
};
```

```
class BatteryOperatedToy2 : virtual public Toy2
{
        friend void repair_toy2(BatteryOperatedToy2);

        private:
                string battery_type;
                int number_of_batteries;
                bool batteries_installed;
                bool in_use;

        protected:
                virtual bool is_ready_for_playing();
                bool is_being_played_with();

        public:
                BatteryOperatedToy2(string, int, string, int);
                BatteryOperatedToy2(BatteryOperatedToy2&);
                void start_playing();
                void stop_playing();
                void put_batteries(string, int);
                void take_out_all_batteries();
};
```

Observe the `virtual` keyword here while creating the `FlyingToy` and `BattryOperatedToy2` classes

Virtual Inheritance in C++

Let us say both *Parent*$_1$ and *Parent*$_2$, derive from a class called *GrandParent*

- Also assume that there are some inheritable elements in *GrandParent*, which can "travel" to *Child*

Clearly, this is the Diamond Problem – there will be two copies of these elements in *Child* …

- … one via *Parent*$_1$ and another via *Parent*$_2$

You are allowed to do this as well, again, as long as you use the scope resolution operator

However, it may be inefficient (and maybe illogical) to have two copies of *GrandParent* members

C++ allows you to inform about this scenario to the compiler …

- … which can then ensure that a single version of these members reach *Child*

To do so, the information must be provided while creating *Parent*$_1$ and *Parent*$_2$ itself

- By adding the keyword `virtual` before *GrandParent*, you provide the necessary information
- **If you do so, only a single copy of *GrandParent* members are maintained for an object of type *Child***

Constructors and Virtual Inheritance

If you use virtual inheritance as discussed, you *may* have to do something more as well

- If the inheritance is not virtual, the job of calling the *GrandParent*'s constructor is with the Parents
- However, if the inheritance is virtual, the usual flow of constructors do not happen

In case of virtual inheritance, the *GrandParent*'s constructor cannot be called implicitly …

- … with the exception of a case where *GrandParent* has a default constructor that can be invoked automatically

Thus, this call must also be added to the list of constructor calls made in the *Child*'s constructor, e.g.

```
PlaneToy::PlaneToy(string s1, int i1, string s2, int i2, string s3)
        : FlyingToy(s1, i1, 2), BatteryOperatedToy2(s1, i1, s2, i2), Toy2(s1, i1)
```

- Observe the call to `Toy2` at the end …

The Complete Example

```
int main()
{
        PlaneToy pt("Aeroplane", 999, "AA", 3, "Airbus A380");
        pt.put_batteries("AA", 3);
        pt.start_playing();
        pt.stop_playing();
}
```

```
Creating a new Aeroplane with id 1
It is a flying toy :-)
It is a battery-operated toy with the requirement of 3 batteries of type AA
You are now looking at a plane toy of type Airbus A380
Batteries Installed !!
Let's play with a model plane of type Airbus A380
Isn't it fun to see something fly like that?
Switching on the toy...
You are now playing with Aeroplane with id 1
Looks like you are feeling dizzy with all that "flying"!
Switching off the toy...
The Aeroplane with id 1 is lying idle
You don't like planes??
Destroying the Aeroplane with id 1
```

Check the Homework !! :D

Homework !!

Download the complete code for Week 4, and build it (check the `makefile` to figure out how to do so)

- Browse through the different code files in the directory
- Produce the output as shown on the right side of the last slide
- Trace the messages printed on the console to the methods and classes they come from