**Chapter 9: Inheritance**

Table of Contents

[Access Modifiers 4](#_Toc49595372)

[Overriding 6](#_Toc49595373)

[Multilevel Inheritance 9](#_Toc49595374)

[Multiple Inheritance 11](#_Toc49595375)

[Aggregation 13](#_Toc49595376)

We have previously touched upon the theory of inheritance in the introductory lecture. Essentially, we can create a class that inherits attributes and functions from another class. The new class is called the child, derived or sub class, while the class it inherits from is called the parent or super class. If a class has no parents, it can also be called the base class.

For example, consider the Counter class below:

class Counter  
{  
protected:  
 int count;  
public:  
 Counter(): count(0){}  
 Counter(int i): count(i){}  
 void incCount(){count++}  
 int getCount(){return count;}  
};

C++

We can create another class that inherits from the Counter class like this:

class CounterChild : public Counter  
{  
public:  
 void decCount(){count--;}  
};

C++

There are a few things to unpack here, but first, let us look at the basics of inheritance. Say we call these classes from the main function like this:

int main()  
{  
 CounterChild c1;  
 c1.incCount();  
 c1.decCount();  
 cout<<c1.getCount();  
}

C++

This code will work. Here, we used the getCount function, which was part of the Counter class, not the CounterChild class, but the function can still be called. This is due to inheritance. We also used the decCount function, which was part of the CounterChild class, but that function decrements the integer count which is again, part of the Counter superclass.

## Access Modifiers

Notice that we used a new keyword, protected, and placed the integer count under it. This is very important. protected is another access modifier, like private and public. Items that are declared public are accessible from anywhere in the program, including from the class, any of its subclasses, or completely separate functions. private items are only accessible from within the class, not even from its subclasses let alone any other functions. protected items are accessible from within the class and from any subclasses, but not from any other functions. We had to declare the integer count as protected since we used it in the decCount function in the CounterChild subclass. Had we declared it as private like we normally do, we would be unable to do this.

Another thing to notice is how we declared the subclass:

class CounterChild: public Counter

C++

We are creating a new class called CounterChild, and telling it to inherit from the existing Counter class. Obviously, we need to have created the Counter class before doing this. We are also using the keyword public. This keyword is the type of inheritance, which defines how the attributes and functions we inherit will behave within the CounterChild class.

Since we declared it to be public, the functions and data we inherit will be available for use from outside the class. For example, we called incCount on an object of the CounterChild class in the main function. Note that this cannot force things with a higher level of security, like protected and private to become public. The protected integer count is not going to become accessible from the main function. We do not even need to consider private things, because CounterChild cannot access them in the first place.

If we had declared it protected, everything we inherit would remain inside the CounterChild class and any further subclasses only. We would be unable to call the incCount or getCount functions as we did from the main function, but we would still be able to use the decCount function since that is declared as public in the CounterChild class.

If we had declared them as private, the same thing would happen, but any further subclasses of the CounterChild class would be unable to access those attributes.

Note that regardless of the type of inheritance, the superclass is unaffected. Even if we inherit from the Counter class privately, if we created an object of the Counter class, we would still be able to use all its functions through that object.

## Overriding

Overriding is another part of polymorphism. Essentially, this means that when a function is defined in both the superclass and the subclass, objects of the subclass will use the function defined in the subclass. For example, say we have a superclass Teacher and a subclass Professor, and we have a getName function in each of them. The getName function from the Teacher class just returns the name but the getName function from the Professor class attaches the word ‘Professor’ to the name before returning it. Objects of the Professor class would return things like ‘Professor কুদ্দুস মিয়া’, but objects of the Teacher class would return things like ‘শুধু মিয়া’. Similarly, if we had another subclass Lecturer that does not have a getName function defined within it, it will use the function from the Teacher class.

There is a specific way we could do this. Inside the Professor class, we could have a function like this:

string getName()  
{  
 return "Professor" + Teacher::getName();  
}

C++

The scope resolution operator is used to call the getName function of the Teacher class, the result is concatenated with "Professor", and then returned by the function. In this way, we can access functions of parent classes that have the same name inside the derived class.

Constructor functions are interesting though. By default, if we have a constructor function in the parent class but not in the derived class, the compiler will use the constructor function in the parent class for objects of the derived class. If we have constructor functions in both classes, both would run. We can also force this behaviour by defining a constructor function in the derived class like this:

CounterChild() : Counter() {}

C++

This calls the constructor function for CounterChild, which in turn uses the initializer list to call the constructor function for Counter. Note that we could not have called the constructor function of Counter inside the braces. It would cause a compilation error. We could have added more code to run inside the braces, which would run after the constructor function of Counter. This makes sense. When we create an object of the CounterChild class, we want things in that class and its parent class to be initialized before we begin our work. Again, if we had a one argument constructor function in the parent class that we wanted to call from the child class, we would create a constructor function like this:

CounterChild (int c) : Counter(c) {}

C++

If we do not have a zero-argument constructor function in the parent class, then we would be forced to do this. Alternatively, we could pass a default value to the parent class constructor and make the derived class constructor a zero-argument constructor.

CounterChild () : Counter(0) {}

C++

From the main function, when we create an object of the CounterChild class, we need only be concerned with how the constructor function of the CounterChild class will work. That constructor will handle the constructor function of the Counter class. We do not need to think about that in the main function.

## Multilevel Inheritance

We can create a class that inherits from a derived class in the same manner. This third class will also contain the properties of the original base class. This is called multilevel inheritance. We could have something like this:

class Iutian  
{  
 int ID;  
 string name;  
}  
  
class UG\_Iutian : public Iutian  
{  
 int semester;  
}  
  
class CSE\_UG\_Iutian : public UG\_Iutian  
{}

C++

Notice that the CSE\_UG\_Iutian class does not have anything new in it. It is essentially the same as the UG\_Iutian class. The difference is in the concept. Say we have some idea that we could have new data soon that applies specifically to the CSE department. We could easily place this data inside the CSE\_UG\_Iutian class. If we had done this in the UG\_Iutian class directly, we would have had to edit data for all objects of the class, which also included objects outside the CSE department. Similarly, if we had some new data that applied to all Iutians, we would add it to just the Iutian class to see its effect on all objects of derived classes as well.

Sometimes, there will be situations where the actual base class will have no objects of its own. Its sole purpose will be to provide a base from which other classes are derived. Such a class is called an abstract class.

In the last lecture we talked about how the constructor functions of super classes and subclasses worked. In multilevel inheritance, things work the same way, but the third level class is not concerned with the first level class. For example, the constructor function of the CSE\_UG\_Iutian class could look like this:

CSE\_UG\_Iutian : UG\_Iutian()  
{  
  
}

C++

We do not call the constructor function of the Iutian class here. That is being dealt with by the constructor function of the UG\_Iutian class.

## Multiple Inheritance

We can inherit from multiple classes like this:

class Iutian  
{  
 int id;  
}  
  
class Cadet  
{  
 int id;  
}  
  
class IutianCadet : public Iutian, public Cadet  
{}

C++

Notice that both the Iutian class and the Cadet class have a variable named id. If we want to write a function in the IutianCadet class that accesses the element id of the Iutian class, we could do it using the scope resolution operator like this:

Iutian :: id = 1;

C++

Given that things are public, we could do the same using an object of the IutianCadet class in the main function like this:

IutianCadet someone;  
someone.Iutian :: id = 1;

C++

Normally though, we just use different names for objects to avoid this mess.

There is a potential problem to multiple inheritance. Say we have two classes B and C, that inherit from class A. Class Bwould have one copy of the data from class A, and class C would have the same copy of data from class A. If we create a new class D that inherits from class B and class C, it would attempt to inherit two copies of the same data that belonged to class A. This is called diamond shaped inheritance and will cause a compilation error if we tried to use data from the class A.

## Aggregation

Normally, a class inherits from another. This is called an ‘is a’ relationship, as in, UG\_Iutian is a(n) Iutian. We have another property related to inheritance, aggregation, that is called the ‘has a’ relation, as in, University has a Student. Here, a class contains objects from another class. This is similar to nested structures in that we have objects of one class inside objects of another class.

class University  
{  
 Student s[10];  
 Teacher t[2];  
 Staff f[5];  
}

C++

In the main function, we could have something like this:

University IUT;  
IUT.s[0].setName = "Moody Boy";

C++

We created an object, and then accessed a function for another object from a different class inside that object.

Composition is similar to aggregation, but is also limited to lifetimes. An object of the Door class lives and dies with an object of the Car class. This is more of a theoretical difference than a programming one.