Lesson 13: Inheritance

13.1. Introduction

In our previous lesson, we learnt about classes and objects. You learnt about object oriented analysis, object oriented design and object oriented implementation. You learnt how to bind together in units called objects using the principle of encapsulation. Using the principle of data abstraction, the members of the class were declared by providing essential information such as member name, data type and visibility. In this lesson, you will learn about another principle of object oriented programming called inheritance.

13.2. Lesson objectives

By the end of this lesson the learner will be able to:

- Describe the difference between base class and derived class.
  Describe the importance of inheritance.
- Discuss different types of inheritance
- Write simple programs in c++ to demonstrate the principle of inheritance.

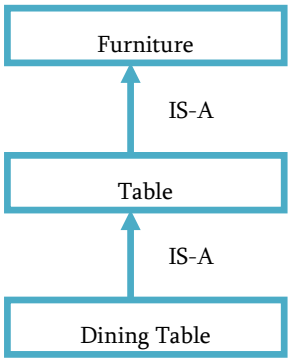
13.3. Lesson outline

This lesson is organized as follows:

13.4. Inheritance in classes

Inheritance is an object oriented programming principle which involves creating a new type that extends the characteristics of an existing type. Inheritance makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class**, and the new class is referred to as the **derived class**. The idea of inheritance implements the is a relationship. For

example, table IS-A furniture, dining table IS-A table hence dining table IS-furniture as well and so on.



13.5. Base class versus Derived Class

A class can be derived from one or more classes. This 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 can be 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. Consider a base class Shape and its derived class Rectangle as follows:

```
#include <iostream>
 using namespace std;
class Shape // Base class
{
    protected:
          int width;
          int height;
    public:
    void setWidth()
    {
     cout<<"Enter the width:"<<endl;
     cin>>width;
    }
    void setHeight()
    {
     cout<<"Enter the height:"<<endl;
     cin>>height;
    }
                Shape(){};
```

```
            ~Shape(){};
 };
class Rectangle: public Shape // Derived class
{   public:
        int getArea()
        {
          return (width * height);
        }
            Rectangle(){};
            ~Rectangle(){};
    };
    void main()
    {
      Rectangle Rect;
      Rect.setWidth();
      Rect.setHeight();
      cout << "Total area: " << Rect.getArea() << endl;
    }
```

13.6. Access control and inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared **private** in the base class. The table below summarizes different access types according to who can access them.

Access	public	protected	private
Same class	yes	Yes	yes
Derived classes	yes	Yes	no
Outside classes	yes	No	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

13.7. Types of inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the accessspecifier as explained above.

We hardly use **protected** or **private** inheritance but **public** inheritance is commonly used.

While using different type of inheritance, following rules are applied:

[a]. **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

[b]. **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

[c]. **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

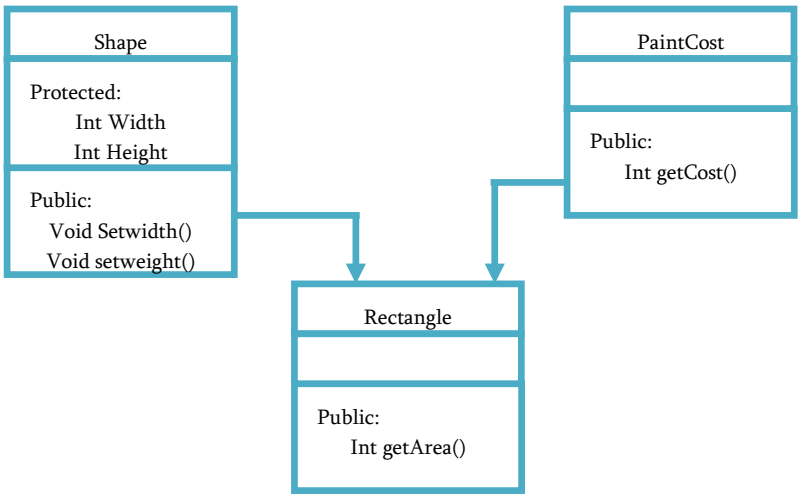
13.8. Multiple inheritances

A C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

Example1: Write a program with three classes shape, rectangle and paint cost to show multiple inheritances. Rectangle inherits from two base classes, paint cost and shape.

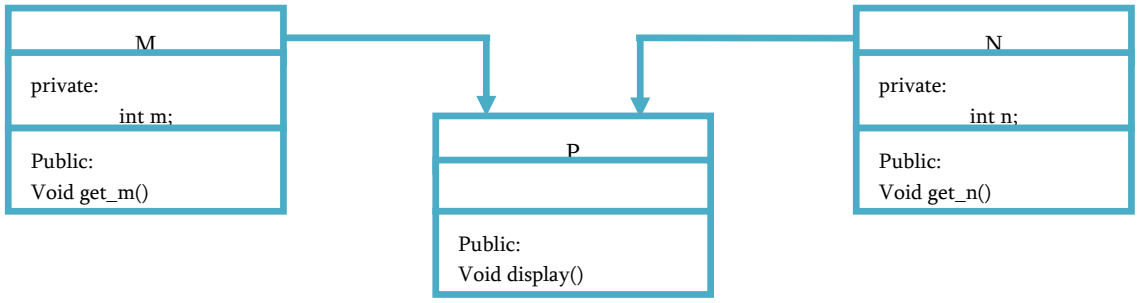



```
#include <iostream>
```

```
using namespace std;
class Shape // Base class Shape
{
 protected:
          int width;
          int height;
public:
    void setWidth()
    {
      cout<<"Enter the width:"<<endl;
      cin>>width;
    }
    void setHeight()
    {
      cout<<"Enter the height:"<<endl;
      cin>>height;
    }
Shape(){};
 ~Shape(){};
};
class PaintCost // Base class PaintCost
{   public:
    int getCost(int area)
    {
      return area *70;
    }
                PaintCost(){};
                ~PaintCost(){};
};
class Rectangle: public Shape, public PaintCost // Derived class
{   public:
         int getArea()
         {
           return (width * height);
         }
                Rectangle(){};
               ~Rectangle(){};
    };
    void main()
    {
  Rectangle Rect;
 int area; //declare local variable
  Rect.setWidth();
  Rect.setHeight();
```

```
  area = Rect.getArea(); //call public function
  cout << "Total area: " << Rect.getArea() << endl;//print area
  cout << "Total paint cost: Kshs" << Rect.getCost(area) << endl;//print cost
}
```

Example 2: In the program below, class P inherits from two base classes- class M and N




```
#include<iostream>
using namespace std;
class M //create a base class M with its members
{
protected:
        int m;  public :
        void get_m();
};
class N  //create a base class N with its members
{
protected:
        int n;  public:
        void get_n();
};
    class P: public M, public N //create a derived class P to inherit from M and N
    {
    public:
            void display(void);
    };
    void M ::get_m() //define member function for M
    {
            cout<<"Enter value of m"<<endl;
            cin>>m;
    }
    void N::get_n()//define member function for N
    {
        cout<<"Enter value of n"<<endl;
        cin>>n;
    }
```

```
void P:: display(void) //define member function for P
{
     cout<<"m="<<m<<endl;
     cout<<"n="<<n<<endl;
    cout<<"m*n="<<m*n<<endl;
}
void main() //create interface
{
 P p;
          p.get_m();
          p.get_n();
          p.display();
 }
```

13.9. Advantages of inheritance

Creating new classes from existing ones provides a number of benefits.

- It saves time because much of the code needed for the new class is already written.
- It saves additional time because the existing code has already been tested—that is, you know it works correctly; it is reliable.
- More time is saved because you already understand how the base class works, and you can concentrate on the complexity added as a result of your extensions to the class.
- In a derived class, you can extend and revise a parent class without corrupting the existing parent class features. In other words, you don't have to modify a parent class to get it to work correctly with your new category of objects; you leave the original class alone.
- If other classes have been derived from the parent class, the parent class is even more reliable—the more its code has been used, the more likely it is that logical errors have already been found and fixed.

13.10. Disadvantages of inheritance

- Developing good, general-purpose software from which more specific classes can be inherited is more difficult and time-consuming.
- Multiple inheritances present a limitation since the definition of a class that inherits from two or more base classes is hard to understand and more prone to errors.

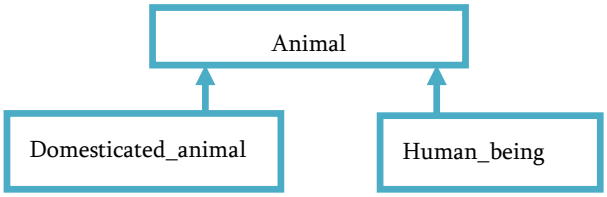
13.11. Revision questions

[a]. Describe two advantages and two disadvantages of inheritance.

[b]. Briefly discuss how each of the following visibility modifiers affects inheritance of member elements in derived classes:

[i]. Protected
[ii]. Private
[iii]. Public

[c]. Write a program with base class animal with features, which can be inherited by class Human being and class domesticated animal. Create instances from all classes to demonstrate aspect of inheritance. Use the diagram below



13.12. Summary

In this lesson, you have learnt about inheritance. You have learnt how to create base classes and derive new classes from the base classes. You have also learnt how to write programs with single and multiple inheritances. Finally you have learnt about advantages and disadvantages of inheritance.

13.13. Suggested reading

[1]. Sams Teach Yourself C++ in 24 hours by Jesse Liberty and Rogers Cadenhead.
[2]. Object oriented programming using c++ by Joyce Farrell 4th.Edition
[3]. Object oriented programming with C++ by E Balagurusamy 3rd ed; publisher: Tata Mcraw Hill
[4]. Object-oriented programming with c++ by Sourav Sahay.