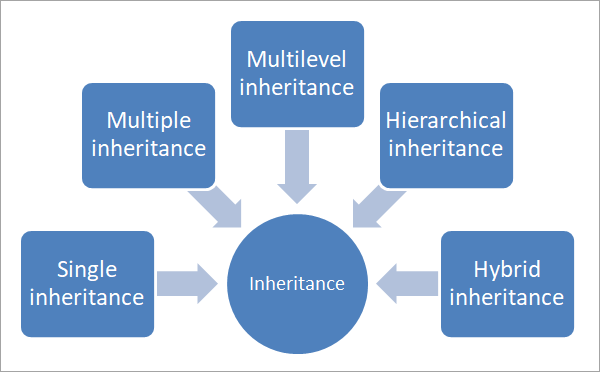
**OBJECT ORIENTED PROGRAMMING (PCC-CS503)**

**Unit – 5**

**Essentials of Object Oriented Programming**

Inheritance – Types of Inheritance



1) **Single Inheritance**: In single inheritance, a class derives from one base class only. This means that there is only one child class that is derived from one base class.

**Syntax:**

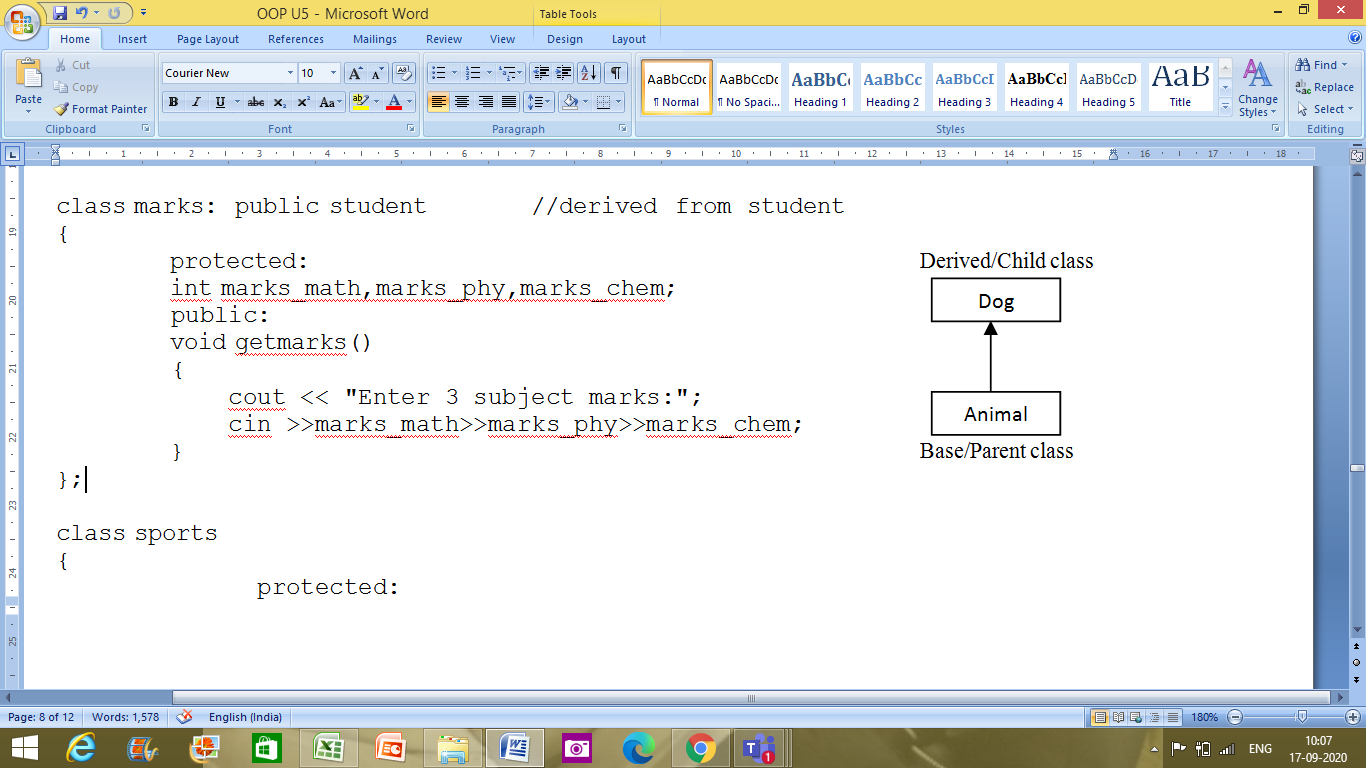
class derivedclass : visibilitymode baseclass

{

//class specific code;

};

**Example of Single Inheritance**



|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class Animal  {     string name="";     public:     int tail=1;     int legs=4;    };  class Dog : public Animal  {     public:     void voiceAction()     {        cout<<"Barks!!!";     }  };  int main()  {     Dog dog;     cout<<"Dog has "<<dog.legs<<" legs"<<endl;     cout<<"Dog has "<<dog.tail<<" tail"<<endl;     cout<<"Dog ";     dog.voiceAction();  } |

**Output:**

Dog has 4 legs  
Dog has 1 tail  
Dog Barks!!!

We have a class Animal as a base class from which we have derived a subclass dog. Class dog inherits all the members of Animal class and can be extended to include its own properties, as seen from the output.

2) Multiple Inheritance: Multiple inheritance is a type of inheritance in which a class derives from more than one classes. As shown in the below diagram, class C is a subclass that has class A and class B as its parent.

**Syntax:**

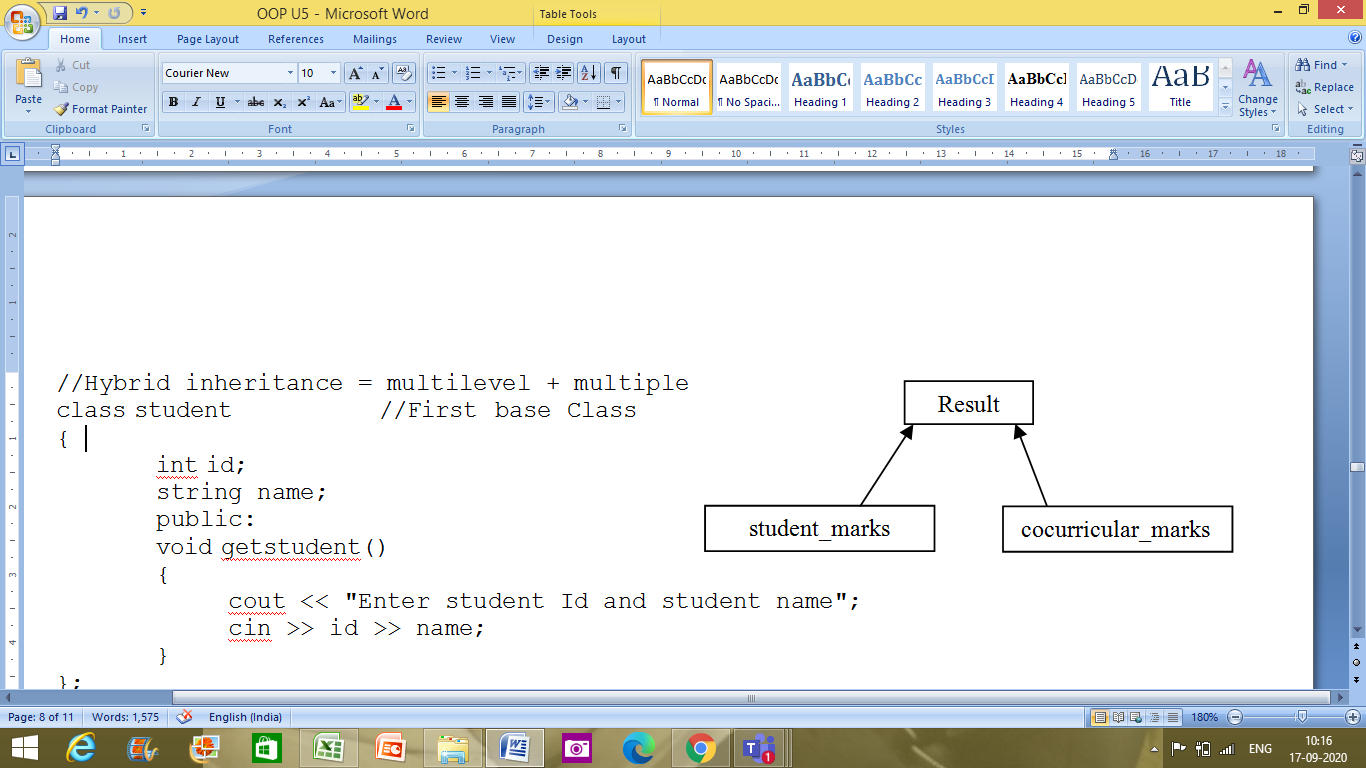
class derivedclass : visibilitymode baseclass1, visibilitymode baseclass2

{

//class specific code;

};

**Example of Multiple Inheritance**



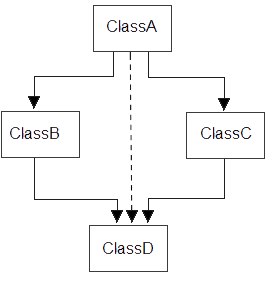
|  |
| --- |
| #include <iostream>  using namespace std;  //multiple inheritance example  class student\_marks  {  protected:  int rollNo, marks1, marks2;  public:  void get()  {  cout << "Enter the Roll No.: ";  cin >> rollNo;  cout << "Enter the two highest marks: ";  cin >> marks1 >> marks2;  }  };  class cocurricular\_marks  {  protected:  int comarks;  public:  void getsm()  {  cout << "Enter the mark for CoCurricular Activities: ";  cin >> comarks;  }  };    //Result is a combination of subject\_marks and cocurricular activities marks  class Result : public student\_marks, public cocurricular\_marks  {     int total\_marks, avg\_marks;     public:     void display()     {        total\_marks = (marks1 + marks2 + comarks);        avg\_marks = total\_marks / 3;        cout << "\nRoll No: " << rollNo << "\nTotal marks: " << total\_marks;        cout << "\nAverage marks: " << avg\_marks;     }  };  int main()  {  Result res;  res.get(); //read subject marks  res.getsm(); //read cocurricular activities marks  res.display(); //display the total marks and average marks  } |
|  |

**Output:**

Enter the Roll No.: 25  
Enter the two highest marks: 40 50  
Enter the mark for CoCurricular Activities: 30

Roll No: 25  
Total marks: 120  
Average marks: 40

**Diamond problem/Ambiguity problem in case of Multiple Inheritance:** Ambiguity in C++ occur when a derived class have two base classes and these two base classes have one common base class. Consider the followling figure:



#include<iostream.h>

#include<conio.h>

class ClassA

{

public:

int a;

};

class ClassB : public ClassA //a,b

{

public:

int b;

};

class ClassC : public ClassA //a,c

{

public:

int c;

};

class ClassD : public ClassB, public ClassC //b,c,a,a,d

{

public:

int d;

};

void main()

{

ClassD obj;

//obj.a = 10; **//Statement 1, Error occur**

//obj.a = 100; **//Statement 2, Error occur**

obj.ClassB::a = 10; **//Statement 3**

obj.ClassC::a = 100; **//Statement 4**

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;

}

Output :

A from ClassB : 10

A from ClassC : 100

B : 20

C : 30

D : 40

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**, bco'z compiler can't differentiate between two copies of **ClassA** in **ClassD**.

**How ambiguity can be avoided?**

We can resolve this problem by making the root base class as virtual. To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB** and **ClassC** as **virtual** class.

### Example to avoid ambiguity by making base class as a virtual base class

#include<iostream.h>

#include<conio.h>

class ClassA

{

public:

int a;

};

class ClassB : **virtual** public ClassA //a,b

{

public:

int b;

};

class ClassC : **virtual** public ClassA //a,c

{

public:

int c;

};

class ClassD : public ClassB, public ClassC //b,c,a,d

{

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;

cout<< "\n B : "<< obj.b;

cout<< "\n C : "<< obj.c;

cout<< "\n D : "<< obj.d;

}

Output :

A : 100

B : 20

C : 30

D : 40

According to the above example, **ClassD** have only one copy of **ClassA** therefore statement 4 will overwrite the value of **a**, given at statement 3.

3) Multilevel Inheritance: In multilevel inheritance, a class is derived from another derived class. This inheritance can have as many levels as long as our implementation doesn’t go wayward. In the above diagram, class C is derived from Class B. Class B is in turn derived from class A.

**Syntax:**

class Intermediateclass : visibilitymode baseclass

{

//class specific code;

};

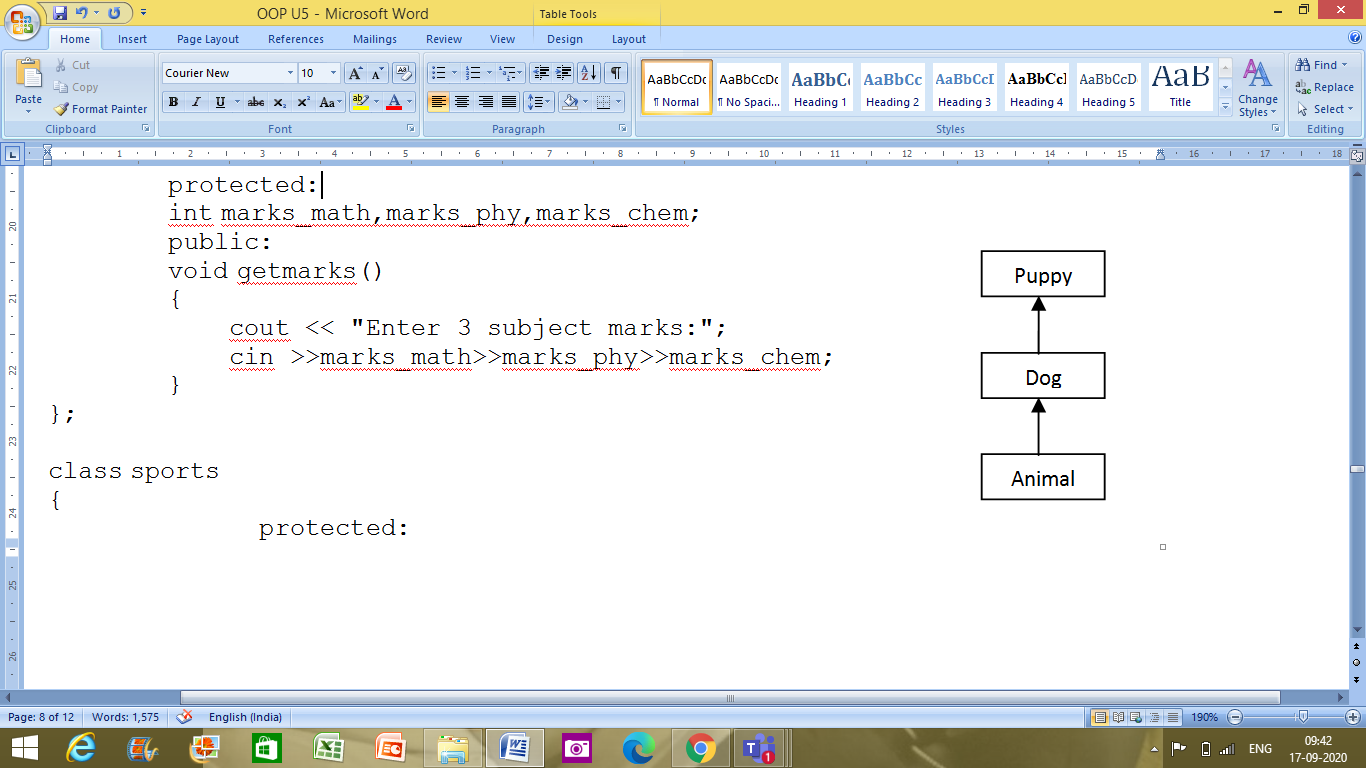
class Derivedclass : visibilitymode Intermediateclass

{

//class specific code;

};

**Example of Multilevel Inheritance**

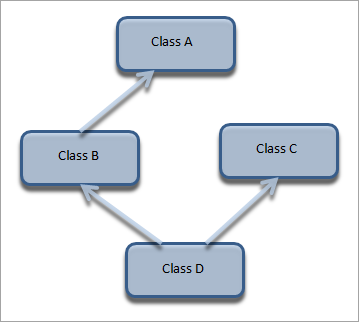


|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class Animal  {     string name="";     public:     int tail=1;     int legs=4;    };  class Dog : public Animal  {     public:     void voiceAction()     {        cout<<"Barks!!!";     }  };  class Puppy:public Dog  {     public:     void weeping()     {        cout<<"Weeps!!";     }  };  int main()  {  Puppy puppy;  cout<<"Puppy has "<<puppy.legs<<" legs"<<endl;  cout<<"Puppy has "<<puppy.tail<<" tail"<<endl;  cout<<"Puppy ";  puppy.voiceAction();  cout<<" Puppy ";  puppy.weeping();  } |

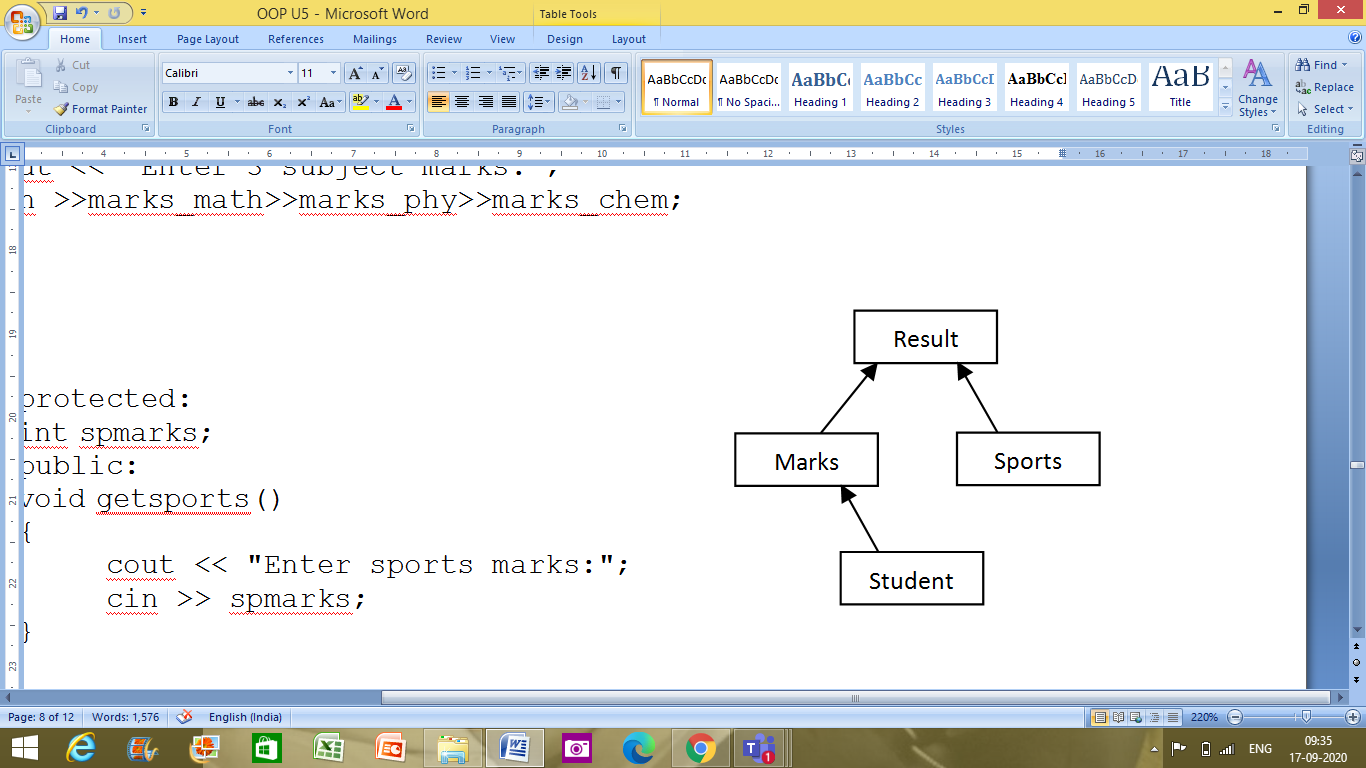
**Output:**

Puppy has 4 legs  
Puppy has 1 tail  
Puppy Barks!!! Puppy Weeps!!

4) Hybrid Inheritance: Hybrid inheritance is usually a combination of more than one type of inheritance. In the above representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B and D) to get a hybrid inheritance.



**Example of Hybrid Inheritance**

****

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  //Hybrid inheritance = multilevel + multiple  class student //First base Class  {         int id;         string name;         public:         void getstudent()  {          cout << "Enter student Id and student name";  cin >> id >> name;      }  };  class marks: public student //derived from student  {  protected:  int marks\_math,marks\_phy,marks\_chem;          public:          void getmarks()  {           cout << "Enter 3 subject marks:";  cin >>marks\_math>>marks\_phy>>marks\_chem;      }  };  class sports  {                protected:                int spmarks;                public:                void getsports()  {                     cout << "Enter sports marks:";  cin >> spmarks;      }  };  class result : public marks, public sports  {  int total\_marks;                  float avg\_marks;                  public :             void display()  {                  total\_marks=marks\_math+marks\_phy+marks\_chem;                  avg\_marks=total\_marks/3.0;                    cout << "Total marks =" << total\_marks << endl;                  cout << "Average marks =" << avg\_marks << endl;                  cout << "Average + Sports marks =" << avg\_marks+spmarks;      }  };    int main(){                  result res;//object//                  res.getstudent();                  res.getmarks();                  res.getsports();                  res.display();                 return 0;  } |
|  |

**Output:**

Enter student Id and student name 25 Ved  
Enter 3 subject marks:89 88 87  
Enter sports marks:40  
Total marks =264  
Average marks =88  
Average + Sports marks =128

Note that in hybrid inheritance as well, the implementation may result in “Diamond Problem” which can be resolved using “virtual” keyword as mentioned previously.

5) Hierarchical Inheritance: In hierarchical inheritance, more than one class inherits from a single base class as shown in the representation above. This gives it a structure of a hierarchy.

**Syntax:**

class Derivedclass\_1 : visibilitymode baseclass

{

//class specific code;

};

class Derivedclass\_2 : visibilitymode baseclass

{

//class specific code;

};

.

.

.

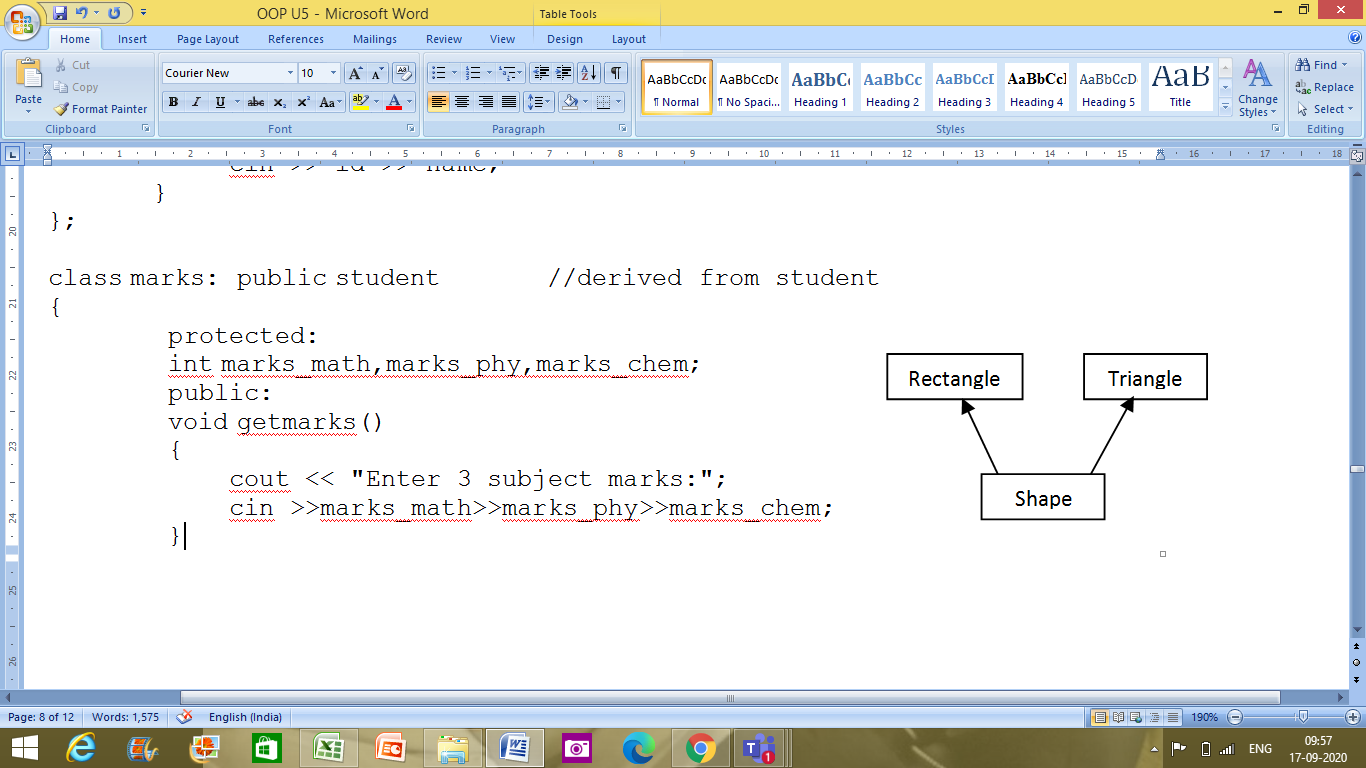
class Derivedclass\_n : visibilitymode baseclass

{

//class specific code;

};

**Example demonstrating Hierarchical Inheritance**



|  |
| --- |
| #include <iostream>  using namespace std;  //hierarchical inheritance example  class Shape                   // shape class -> base class  {  public:  int x,y;    void get\_data(int n,int m)  {        x= n;        y = m;     }  };  class Rectangle : public Shape // inherit Shape class  {  public:  int area\_rect()  {  int area = x\*y;  return area;  }  };  class Triangle : public Shape // inherit Shape class  {  public:  int triangle\_area()  {  float area = 0.5\*x\*y;  return area;  }  };  class Square : public Shape // inherit Shape class  {  public:  int square\_area()  {  float area = 4\*x;  return area;  }  };  int main()  {  Rectangle r;     Triangle t;     Square s;     int length,breadth,base,height,side;    //area of a Rectangle     cout << "Enter the length and breadth of a rectangle: "; cin>>length>>breadth;     r.get\_data(length,breadth);     int rect\_area = r.area\_rect();     cout << "Area of the rectangle = " <<rect\_area<< endl;    //area of a triangle     cout << "Enter the base and height of the triangle: "; cin>>base>>height;     t.get\_data(base,height);     float tri\_area = t.triangle\_area();     cout <<"Area of the triangle = " << tri\_area<< endl;    //area of a Square     cout << "Enter the length of one side of the square: "; cin>>side;     s.get\_data(side,side);     int sq\_area = s.square\_area();     cout <<"Area of the square = " << sq\_area<< endl;     return 0;  } |

**Output:**

Enter the length and breadth of a rectangle: 10 5  
Area of the rectangle = 50  
Enter the base and height of the triangle: 4 8  
Area of the triangle = 16  
Enter the length of one side of the square: 5  
Area of the square = 20

**Operator overloading**

* Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the user-defined meaning to the user-defined data type (classes).
* Operator overloading is used to overload or redefines most of the operators available in C++ and it can be done by creating a user defined **operator function**.
* For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.
* **Operator that cannot be overloaded are as follows:**
  + Scope operator (::)
  + Sizeof
  + member selector(.)
  + member pointer selector(\*)
  + ternary operator(?:)

## Syntax of Operator Overloading

return\_type class\_name :: operator op(argument\_list)

{

     // body of the function.

}

**-**Where the **return type** is the type of value returned by the function.

**-class\_name** is the name of the class.

**-operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

## Rules for Operator Overloading

* Only existing operators can only be overloaded and new operators cannot be formed and overloaded.
* The overloaded operator contains at least one operand of the user-defined data type.
* When unary operators are overloaded through a member function then no explicit arguments are accepted.
* When binary operators are overloaded through a member function then only one explicit argument is accepted.
* Retain the meaning of the operator.
* Retain the syntax of the operator.
* Retain the hierarchy of the operator.

## C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

#include <iostream>

using namespace std;

class Test

{

private:

int num;

public:

Test()

{

num= 8;

}

void operator ++()

{

num = num+2;

}

void Print()

{

cout<<"The Count is: "<<num;

}

};

int main()

{

Test tt;

++tt; // calling of a function "void operator ++()"

tt.Print();

return 0;

}

**Output:**

The Count is: 10

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

#include <iostream>

using namespace std;

class A

{

int x;

public:

A(int i)

{

x=i;

}

void operator+(A);

};

void A :: operator+(A a)

{

int m = x+a.x;

cout<<"The result of the addition of two objects is: "<<m;

}

int main()

{

A a1(5);

A a2(4);

a1+a2; //a1.operator+a2 :

return 0;

}

**Output:**

The result of the addition of two objects is: 9

**NOTE:** On execution of the following statement:

a1+a2; //a1.operator+a2

1. The object a1 (which is on the left side of + operator) generates a call (or invokes) to the operator function.
2. The object a2 (which is on the right side of + operator) is passed explicitly as an argument to the operator function.

**Pointers to Objects**

**Assignment of an Object to another Object**