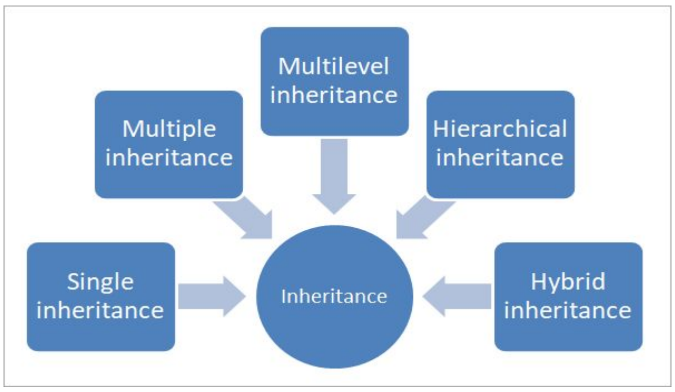
OBJECT ORIENTED PROGRAMMING (PCC-CS503)

Unit – 5

Essentials of Object Oriented Programming

Inheritance – Types of Inheritance

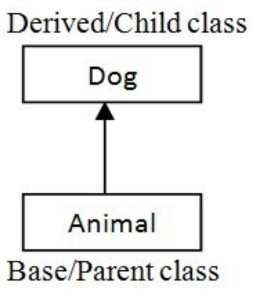


1) **Single Inheritance**: In single inheritance, a class derives from one base class only. This means that there is only one child class that is derived from one base class.

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!!!
```

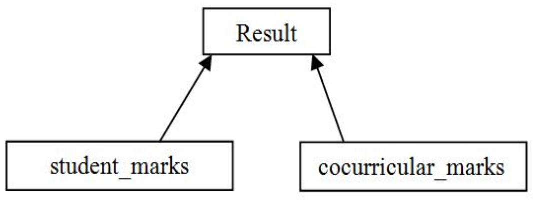
We have a class Animal as a base class from which we have derived a subclass dog. Class dog inherits all the members of Animal class and can be extended to include its own properties, as seen from the output.

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;
};
```

<u>Example of Multiple Inheritance</u>



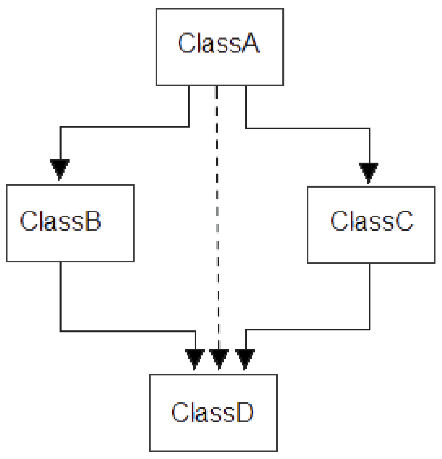
```
#include <iostream>
using namespace std;

//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
```

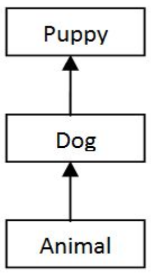
According to the above example, **ClassD** have only one copy of **ClassA** therefore statement 4 will overwrite the value of **a**, given at statement 3.

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;
};
```

<u>Example of Multilevel Inheritance</u>



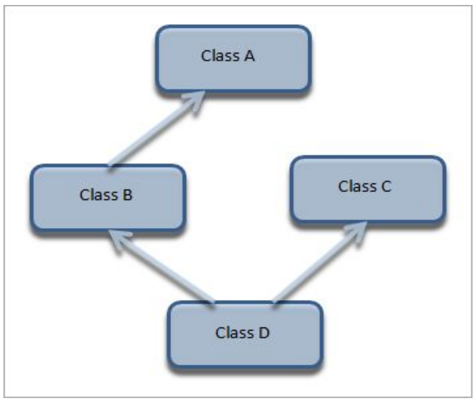
```
#include <iostream>
#include <string>

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();
}
```

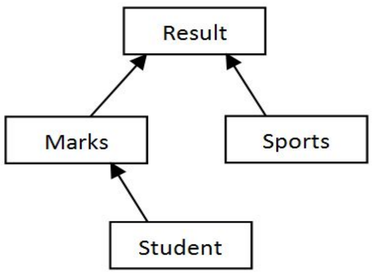
Output:

```
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
```

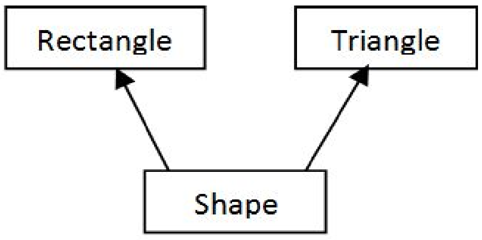
Note that in hybrid inheritance as well, the implementation may result in “Diamond Problem” which can be resolved using “virtual” keyword as mentioned previously.

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;
};
```

<u>Example demonstrating Hierarchical Inheritance</u>



```
#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 no 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 ++()" num:tt =10
       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);  //x:a1=5
    A a2(4);  //x: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

- We use *pointers* to point to the variables of primitive data types like int, char, float etc.,
- we can also use pointers to objects as objects can also have have their address

Example of Pointers to object:

```
#include <iostream>
using namespace std;

 class ptrobj
 {
    int i;
    public:
        void read(int j)
        {
            i=j;
        }
        int display()
        {
            return i;
        }
 };

int main()
{
    ptrobj obj;
    ptrobj *pobj;
    pobj = &obj; // get address of obj
    pobj->read(10);              // use -> to call read()
    cout<<pobj->display();       // use -> to call display()
    return 0;
}
```

O/P:

```
10
```

Assignment of an Object to another Object

When one object's value is assigned to another object, the first object is copied to the second object. Therefore,

```
Point a, b;
...
a = b;
```
//causes the value of *b* to be copied to *a*.

This can be done in two ways:
- by using assignment operator
- by using copy constructor

Example:

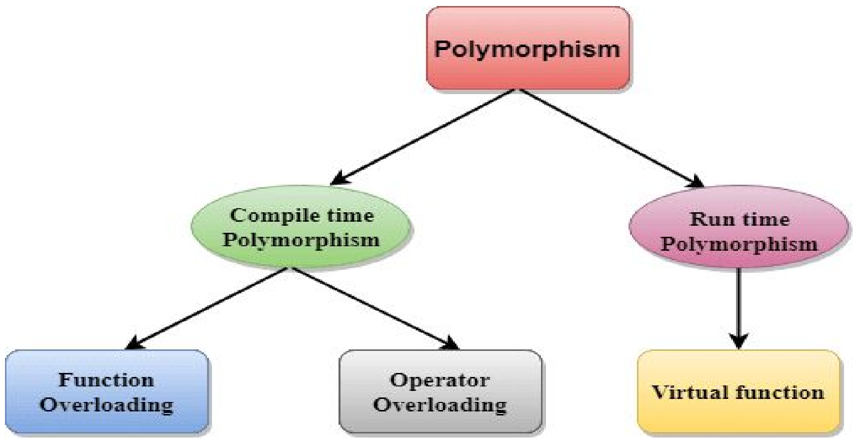
```
class MyClass
{
         int a, b;
         public:
                   void setAB(int i, int j)
                    {
                       a = i, b = j;
                    }
                   void display()
                   {
                       cout << "\n a is " <<a << "\n";
                       cout << "\n b is " << b <<"\n";
                    }
 };

void main()
 {
      clrscr();
      MyClass ob1, ob2;
      ob1.setAB(10, 20);
      ob2.setAB(0, 0);
      cout << "ob1 before assignment";
      ob1.display();
      cout << "ob2 before assignment";
      ob2.display();

      ob2 = ob1;  //assignment of an obj to another obj

      cout << "ob1 after assignment";
      ob1.display();
      cout << "ob2 after assignment";
      ob2.display();

      ob1.setAB(-1, -1);
      cout << "ob1 after changing ob1 :";
      ob1.display();

      cout << "ob2 after changing ob1 :";
      ob2.display();
      getch();
 }
```

O/P:

```
ob1 before assignment
 a is 10
 b is 20
ob2 before assignment
 a is 0
 b is 0
ob1 after assignment
 a is 10
 b is 20
ob2 after assignment
 a is 10
 b is 20
ob1 after changing ob1 :
 a is -1
 b is -1
ob2 after changing ob1 :
 a is 10
 b is 20
```

Polymorphism through dynamic binding

- Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms.
- A simple code/operator/function may behave differently in different situations.
- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.
- Polymorphism is of two types –
    - Compile time polymorphism (demonstrates static/early binding)
        - Function Overloading
        - Operator Overloading
    - Run time polymorphism (demonstrates dynamic/late binding)
        - Function Overriding (Achieved with help of virtual keyword)



Compile time polymorphism:

fun(){...};

fun(int a) {...};

fun1() {...};

fun2() {...}

fun(3);

- The binding of the function call with the function definition when happens at compile time, then it is called compile time polymorphism
- Also called also known as static binding or early binding.

- The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.
- It is achieved by function overloading and operator overloading.

  Function Overloading: Whenever same function name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as function overloading.

- Example: Two versions of the same function (with same name but different signatures). Depending on the number of arguments being passed during the call, the appropriate function is called by the program.

  myFunction(int a, int b)
  myFunction(double a, double b)
  There are two ways of overloading a function –
  - Changing number of arguments like myFunction(int a) and myFunction(int a, int b)
  - Changing type of arguments.

  Operator overloading: Operator overloading is possible in C++, you can change the behaviour of an operator like +, -, & etc to do different things.

  - For example, you can use + to:
    - concatenate two strings
    - add two objects
    - add two complex nos.
    - add two int nos.
  - Consider two strings s1 and s2, we can concatenate two strings s1 and s2 by overloading + operator as String s, s = s1 + s2;

  For Example – We know that ++x increments the value of x by 1. What if we want to overload this and increase the value by 100. The program below, does that.

Run time polymorphism:

- The binding of the function call with the function definition when happens at run time, then it is called run time polymorphism
- Also called also known as dynamic binding or late binding.
- In Late Binding function call is resolved at **runtime**. Hence, now compiler determines the type of object at runtime, and then binds the function call.
- It can be achieved by using the concept of method overriding.
- **Function overriding**, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes.

Example:

```
#include <iostream>
using namespace std;
```

```
class Animal
{
    public:
      void eat()
      {
            cout<<"Eating...";
      }
};

class Dog: public Animal
{
      public:
        void eat()
        {
           cout<<"Eating bread...";
        }
};

int main(void)
{
   Dog d = Dog();
   d.eat();
   return 0;
}
```

O/P:

```
Eating bread...
```

NOTE: In the above case, the prototype of eat() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

There are two ways run time polymorphism may be achieved in C++

- Function Overriding
- Virtual Functions (Solves problem of static resolution while working with pointers)

Compile time polymorphism	**Run time polymorphism**
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.

It is achieved by function overloading and operator overloading.	It is achieved by function overriding and virtual functions.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

1. Use the concept of Operator overloading to overload '-' operator in order to subtract two complex numbers.

2. Use the concept of Operator overloading to redefine the use of '++' so that when this is encountered, then the value of variable increases by 100 and not by 1.

++x □ x=x+100

3. WAP that accepts two objects of time class in hh:mm:ss format and is finally add the two times and display the addition result.

Example: class time: int hrs, int min, int sec
Obj1: 09:22:35
Obj2: 02:45:53
Overload '+' operator to add two objects to get total time.

Virtual Functions

- A virtual function is a member function which is declared within a base class and is redefined (or overridden) in the derived class.
- When you refer to a derived class object using a pointer (or a reference) to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.
- When calling a function using pointers or references, the following rules apply:
    - A call to a virtual function is resolved according to the underlying type of object for which it is called.
    - A call to a non-virtual function is resolved according to the type of the pointer or reference.

Rules for Virtual Functions

- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Example:

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
	virtual void print()
	{
		cout << "print base class" << endl;
	}

	void show()
	{
		cout << "show base class" << endl;
	}
};

class derived : public base {
public:
	void print()
	{
		cout << "print derived class" << endl;
	}

	void show()
	{
		cout << "show derived class" << endl;
	}
};

int main()
{
	base* bptr, *kptr;
	derived d;
	bptr = &d;

	// virtual function, binded at runtime
	bptr->print();

	// Non-virtual function, binded at compile time
	bptr->show();
    }
```

O/P:

```
print derived class
show base class
```

Let’s look at the concept again:

```
int main()
{
	base* bptr, *kptr;
	derived d;
	base b;
```

```
        bptr = &d;
        kptr = &b;

        // virtual function, binded at runtime
        bptr->print();

        // Non-virtual function, binded at compile time
        bptr->show();

        cout<<"when base pointer points to base class:"<< endl;

        // virtual function, binded at runtime
        kptr->print();

        // Non-virtual function, binded at compile time
        kptr->show();
    }
```

O/P:

```
print derived class
show base class
when base pointer points to base class:
print base class
show base class
```

Overloading, overriding and hiding

The technique of overriding base class methods must be used with caution however, to avoid unintentionally **hiding** overloaded methods – a single overriding method in a derived class will **hide** all overloaded methods of that name in the base class.

```
class Base
{
public:
        int fun()
        {
                cout<<"Base::fun() called";
        }
        int fun(int i)
        {
                cout<<"Base::fun(int i) called";
        }
};

class Derived: public Base
{
public:
        int fun()
        {
                cout<<"Derived::fun() called";
        }
};

int main()
{
        Derived d;
        d.fun(); // No error
        d.fun(5); //Compile time error! Base::fun(int i)is hidden
        return 0;
}
```

The parameter list and return type for the member function in the derived class must match those of the member function in the base class. Otherwise, the member function of the derived class **hides** the member function in the base class, and no polymorphic behaviour will occur.

```
#include<iostream>

using namespace std;

class Base
{
public:
        int fun()
        {
                cout<<"Base::fun() called";
        }
        int fun(int i)
        {
                cout<<"Base::fun(int i) called";
        }
};

class Derived: public Base
{
public:
        int fun(char c) // Makes Base::fun() and Base::fun(int ) hidden
        {
                cout<<"Derived::fun(char c) called";
        }
};

int main()
{
        Derived d;
        d.fun(); // Compiler Error
        return 0;
}
```