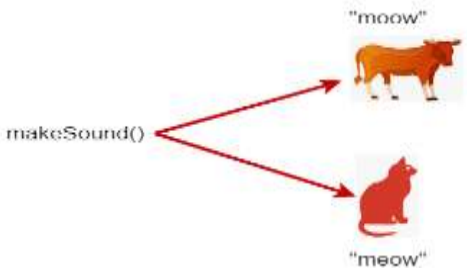
Chapter 4
Polymorphism

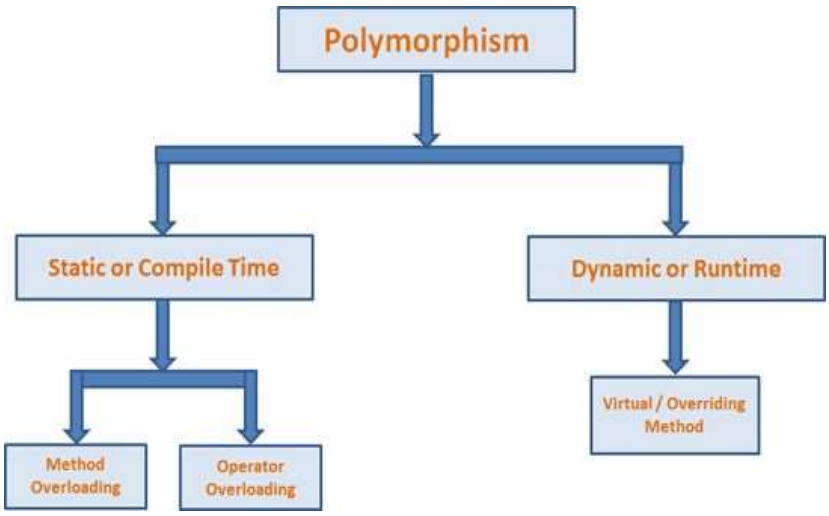
The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered as one of the important features of Object Oriented Programming.

For example, suppose we have the function makeSound(). When a cat calls this function, it will produce the meow sound. When a cow invokes the same function, it will provide the moow sound.



Though we have one function, it behaves differently under different circumstances. The function has many forms.



In C++ polymorphism is mainly divided into two types:

Compile time Polymorphism: This is also known as static (or early) binding.

Runtime Polymorphism: This is also known as dynamic (or late) binding.

1. **Static or Parametric or Compile time polymorphism**

   Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is also called early binding. In this the compiler selects the appropriate function during the compile time.

   This type of polymorphism is achieved by function overloading and operator overloading.

- **Function Overloading**:
  When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.
  Functions can be overloaded by change in number of arguments or/and change in type of arguments.

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

void test(int i) {
        cout << " The int is " << i << endl;
}
void test(float  f) {
        cout << " The float is " << f << endl;
}
void test(char ch[20]) {
        cout << " The string is " << ch << endl;
}
void test(int a,int b)
{
        cout<<"Sum="<<a+b;
}

int main() {
        test(5);
        test(5.5);
        test("five");
        test(10,20);
        return 0;
}
```

Q. WAP to find the area of circle, rectangle and square. (implement function overloading concept).

- **Operator Overloading**:

  The concept by which we can give special meaning to an operator of C++ language is known as operator overloading.

  An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading. Operator overloading allows us to assign multiple meanings to the operators. Compiler generates the appropriate code based on the manner in which the operator is used.

  When an operator is overloaded, that operator does not loses its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.

We can overload the C++ operators except the following:

- ✓ Scope Resolution Operator (: :)
- ✓ Membership Operator (.)
- ✓ Size of operator (size of)
- ✓ Conditional Operator (? :)
- ✓ Pointer to Member Operation (.*)

Rules for Operator Overloading

i. **New operators cannot be created for overloading**

ii. **All operators cannot be overloaded**

iii. **Retain Meaning**

iv. **Retain Syntax**

v. **Retains hierarchy of operators**

vi. **Parameters in overloaded operation function:** Overloaded operator function must either be a member function of the class or a non-member function (friend function). If the overloaded operator function is a member function, then overloaded unary operator takes no parameter, and a binary operator takes one parameter. However, if the overloaded operator function is non-member function, then overloaded unary operator takes one parameter, and a binary operator takes two parameters.

Operator overloading can be done by two ways:

- ➔ Using member function
- ➔ Using friend function

Syntax for operator overloading:

The operator overloading is done by using a special function called operator function, called operator function. The general syntax for operator function is:

```
return_type operator Operator_Symbol(arg list)
{
        //Function body
}
```

Here,

Return_type is the return type of the function.

operator is a keyword.

Operator_symbol is the operator we want to overload. Like: +, <, -, ++, etc.

arguments is the arguments passed to the function.

Example:

a. Overloading Unary Operator (++ (prefix)) // Using operator function or member function

```
#include <iostream>
using namespace std;
class index
{
   public:
   int count;
   void getdata(int i)
   {
      count=i;
   }
   void showdata ()
   {
       cout<< "count=" <<count<<endl;
   }
   void operator ++()
   {
       ++count;
   }
};
int main()
{
   index a1;
   a1.getdata(3);
   a1.showdata();
   ++a1;
```

```
    a1.showdata();
    return 0;
}
```

b. Overloading Unary Operator (++ (postfix)) // Using operator function or member function

```
#include <iostream>
using namespace std;
class index
{
    public:
    int count;
    void getdata(int i)
    {
        count=i;
    }
    void showdata ()
    {
        cout<< "count=" <<count<<endl;
    }
    void operator ++(int dummy)
    {
        count++;
    }
};
int main()
{
    index a1;
    a1.getdata(3);
    a1.showdata();
    a1++;
```

```
    a1.showdata();
    return 0;
}
```

Syntax to use Friend Function in C++ to Overload Operators:

```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
   //Statements;
}
```

```
//Overloading unary increment (++) operator using friend function
#include <iostream>
using namespace std;
class Test
{
   int a;
   public:
      Test()
      {
         a = 0;
      }
      Test(int i)
      {
         a = i;
      }
friend Test operator ++ (Test & op1); //prefix increment
friend Test operator ++ (Test & op1, int dummy); //postfix increment
 void Display();
};
// Overload prefix ++ using a friend function.
  Test operator ++(Test & op1)
{
   ++ op1.a;
   return op1;
```

```
}
 //Overload postfix ++ using a friend function.
  Test operator ++ (Test & op1, int dummy)
{
   op1.a ++;
   return op1;
}
void Test::Display()
{
   cout << a << ", ";
}
int main()
{
   Test a (12);
   a.Display();
   ++a; // prefix increment
   a.Display();
   a++; // postfix increment
   a.Display();
   return 0;
}
```

Overloading Binary Operator

```
// CPP program to illustrate + Operator Overloading to add complex number using member function
#include<iostream>
using namespace std;
class Complex
{
  private:
     int real, imag;
  public:
     Complex(int r , int i)
     {
          real = r;
          imag = i;
     }
 // This is automatically called when '+' is used with between twoComplex objects
```

```
    Complex operator + (Complex &obj)
    {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;
    }
   void print()
   {
      cout << real << " + i" << imag << endl;
    }
};

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
Output:
12 + i9
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

```
// CPP program to illustrate + Operator Overloading to add complex number using friend function
#include<iostream>
using namespace std;
class Complex
{
  private:
      int real, imag;
  public:
      Complex(int r , int i)
      {
            real = r;
            imag = i;
      }
   // This is automatically called when '+' is used with between twoComplex objects
      friend  Complex operator + (Complex obj1, Complex obj2);
       void print()
      {
         cout << real << " + i" << imag << endl;
       }
};
Complex operator + (Complex obj1, Complex obj2)
      {
         Complex res;
         res.real = obj1.real + obj2.real;
         res.imag = obj2.imag + obj2.imag;
         return res;
```

```
    }

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
```

WAP to overload relational operator (<) using member function and friend function.

2. Dynamic or Subtype or Runtime Polymorphism

If a member function is selected while the program is running, then it is called run time polymorphism. This feature makes the program more flexible as a function can be called, depending on the context. This is also called late binding. The example of run time polymorphism is virtual function.

This type of polymorphism is achieved by **Function Overriding**. Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example:

```
#include <iostream>
using namespace std;
 // Base class
class Parent
{
   public:
   void print()
   {
      cout << "The Parent print function was called" << endl;
   }
};
// Derived class
class Child : public Parent
{
   public:
   // definition of a member function already present in Parent
   void print()
   {
      cout << "The child print function was called" << endl;
   }
};
int main()
{
   Parent obj1;    //object of parent class
   Child obj2 ;    //object of child class
   obj1.print();    // obj1 will call the print function in Parent
   obj2.print();    // obj2 will override the print function in Parent and call the print function in Child
   return 0;
}
```

```
The Parent print function was called
The child print function was called
```

Q. Difference between function overloading and function overriding

Basis for Comparison	**Overloading**	**Overriding**
Prototype	Prototype differs as number or type of parameter may differ.	All aspect of prototype must be same.
Keyword	No keyword applied during overloading.	Function which is to be overridden is preceded by keyword 'virtual', in the base class.
Distinguishing factor	Number or type of parameter differs which determines the version of function is being called.	Which class's function is being called by the pointer, is determined by, address of which class's object is assigned to that pointer.
Defining pattern	Function are redefined with same name, but different number and type of parameter.	Function is defined, preceded by a keyword 'virtual' in main class and redefined by derived class without keyword.
Time	Compile time.	Run time.
Constructor/Virtual function	Constructors can be overloaded.	Virtual function can be overridden.
Destructor	Destructor cannot be overloaded.	Destructor can be overridden.
Binding	Overloading achieves early binding.	Overriding refers to late binding.

Example	void area(int a);
	void area(int a, int b);
		Class a
		{
		public:
		virtual void display()
		{ cout << "hello"; }
		};
		Class b:public a
		{
		public:
		void display()
		{ cout << "bye";};
		};

Difference between early and late binding

	Early Binding	**Late Binding**
1	It is also known as compile time polymorphism because compiler selects the appropriate member function for the particular function call at the compile time.	It is also called run time polymorphism because the appropriate member functions are selected while the program is executing or running.
2	The information regarding which function to invoke that matches a particular call is known in advance during compilation. Hence it is also called early binding.	The compiler does not know which function to bind with particular function call until the program is executed.
3	This type of binding is achieved using function and operator overloading.	This type of binding is achieved using virtual function.
4	The function call is linked with particular function at compile time statically. So, it is also called static binding.	The selection of appropriate function is done dynamically at run time. So, it is also called dynamic binding.

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- 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.

Rules for Virtual Functions

1. They must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. 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.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Example: //Run time polymorphism

```
#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;
    derived d;
    bptr = &d;

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

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

Pure Virtual Functions (or deferred method or abstract method):

It is the special case of overriding. It's possible that we'd want to include a virtual function in a base class so that it may be redefined in a derived class, but there is no meaningful definition we could give for the function in the base class. It can be defined in the base class but not implemented. The child class provides its implementation.

```
class Shape
{
  protected:
      int width, height;
  public:
      Shape( int a=0, int b=0)
      {
          width = a; height = b;
      }
      // pure virtual function virtual
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Example:

```
#include <iostream>
using namespace std;
class b
{
  public:
  virtual void show()=0;
};

class Derived: public b
{
  public:
   void show()
      {
          cout<< "inside derived class" ;
      }
};
int main()
```

```
{
   Derived d;
   b *p;
   p = &d;
   p ->show();
}
```

A class that contains pure virtual function is known as pure abstract class.

Difference between run time and compile time polymorphism

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Type conversion /Type casting

Casting happens when an object or a primitive is cast into another object type or primitive type.
For example,
float b = 6; // int gets promoted (cast) to float implicitly
int a =9.99 // float gets demoted to int implicitly
Explicit casting happens when we use C's type-casting expressions

Three types of situations might arise for data conversion between different types:
(i) Conversion form basic type to class type.
(ii) Conversion from class type to basic type.
(iii) Conversion from one class type to another class type.

Conversion form basic type to class type

In this type of conversion the source type is basic type and the destination type is class type. Means basic data type is converted into the class type.

class_object = basic_type_variable;

- For example we have class *employee* and one object of employee *'emp'* and suppose we want to assign the employee code of employee *'emp'* by any integer variable say *'Ecode'* then the statement below is the example of the conversion from basic to class type.

  emp = Ecode ;
  Here the assignment will be done by converting *"Ecode"* which is of basic or primary data type into the class type

The conversion from basic type to the class type is done by **Using constructor**

The constructor in this case takes single argument whose type is to be converted.

```
/* Program to convert basic type to class type using constructor */

#include <iostream.h>
#include <conio.h>
class Time
{
      int hrs,min;
      public:
            Time(int);
                  void display();
};

Time :: Time(int t)  //constructor to convert basic (int) to class (Time)
{
      cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
      hrs=t/60;
      min=t%60;
}

void Time::display()
{
      cout<<hrs<< ":Hours(s)" <<endl;
      cout<<min<< " Minutes" <<endl;
}

void main()
{
      clrscr();

      int duration;
      cout<<"Enter time duration in minutes";
       cin>>duration;

      Time t1=duration;

      t1.display();
      getch();
}
```

Here, we have created an object *"t1"* of class *"Time"* and during the creation we have assigned integer variable *"duration"*. It will pass time duration to the constructor function and assign to the *"hrs"* and *"min"* members of the class *"Time"*

H.W.

WAP to convert float value to feet and inch. (Ex. 2.5 = 2 feet, 6 inches)

Conversion from class type to basic type

In this type of conversion the source type is class type and the destination type is basic type. Means class data type is converted into the basic type.

basic_type_variable = class_object;

For example we have class *Time* and one object of *Time* class *'t'* and suppose we want to assign the total time of object *'t'* to any integer variable say *'duration'* then the statement below is the example of the conversion from class to basic type.

In this case, it is necessary to overload casting operator. Overloaded cast operator does not have return type. Its implicit return type is the type to which object need to be converted.

duration= t ; // where, t is object and duration is of basic data type

Example:

```
/* Program to demonstrate Class type to Basic type conversion. */

#include <iostream.h>
class Time
{
        int hrs,min;
        public:
                Time(int ,int);   // constructor
                operator int();   // casting operator function

};

Time::Time(int a,int b)
{
        cout<<"Constructor called with two parameters..."<<endl;
        hrs=a;
        min=b;
}

Time :: operator int() // overload casting operator
{
        cout<<"Class Type to Basic Type Conversion..."<<endl;
        return(hrs*60+min);
}

void main()
{
        clrscr();
        int h,m,duration;
        cout<<"Enter Hours ";
         cin>>h;
        cout<<"Enter Minutes ";
         cin>>m;
        Time t(h,m);       // construct object
```

```
        duration = t;      // casting conversion
        // OR duration = (int)t
        cout<<"Total Minutes are "<<duration;

    }
```

Conversion from one class type to another class type

In this type of conversion both the type that is source type and the destination type are of class type. Means the source type is of class type and the destination type is also of the class type. In other words, one class data type is converted into another class type.

Object_of_destination_class = Object_of_source_type

This type of conversion can be carried out either by a constructor or an operator function. It depends upon where we want the function to be located- in the source class or in the destination class.

There are two ways to convert from one class type to another class type conversion.

Function in the Source Class

If we want to convert objects of one class to object of another class, it is necessary that the operator function be placed in the source class.

```
class distance
{
    int feet,inch;
    public:
    distance()
    {
        feet=inch=0;
    }
    distance(int f,int i)
    {
        feet=f;
        inch=i;
    }
    void display()
    {
        Cout<<feet<<"ft"<<inch<<"inch"<<endl;
    }
};

class dist
{
    int meter;
    int centimeter;
    public:
    dist(int m, int c)
    {
     meter=m;
    centimeter=c;
    }
    operator distance() //operator function
```

```
        {
                distance d;
                int f, i;
                 f= meter *3.3;
                i=centimeter*0.4;
                f=f+i/12;
                i=i%12;
                return distance(f,i);
        }
};

main()
{
        distance d1;
        dist d2(4,50);
        d1=d2;
        d1.display();
}
```

Function in the Destination class

In this case, it is necessary that the constructor be placed in the destination class. This constructor is a single argument constructor and serves as an instruction for converting the argument's type to the class type of which it is a member.

```
class distance
{
        int meter;
        float cm;
        Public:
        {
                distance(int m, float c)
                {
                        meter=m;
                        cm=c;
                }
        int getmeter()
        {
                return meter;
        }
        float getcm()
        {
                return cm;
        }
};

class dist
{
        int feet,inch;
        public:
```

```
dist()
{
        feet=inch=0;
}
dist(int f,int i)
{
        feet=f;
        inch=i;
}
dist(distance d)
{
int m;
float c;
m=d.getmeter();
c=d.getcm();
feet=m*3.3;
inch=c*0.4;
feet=feet+inch/12;
inch=inch % 12;
}

void display()
{
        cout<<feet<<"ft"<<inch<<"inch"<<endl;
}
};

main()
{
        distance d1(6,40);
        dist d2=d1;
        d2.display();
}
```

Conversion from one class type to another class type

//One class to another class conversion using constructor

```
#include<iostream>
using namespace std;
class Product
{
        int num1,num2;
        public:
                void setData(int x,int y)
                {
                        num1=x;
                        num2=y;
                }
                int getnum1()
                {
                        return num1;
```

```
		}
		int getnum2()
		{
			return num2;
		}
};
class Item
{
	int num3,num4;
	public:
		void showData()
		{
			cout<<"First Value="<<num3<<"Second Value="<<num4;
		}
		Item()
		{
		}
		Item(Product p)  //Conversion Function==> Constructor
		{
			num3=p.getnum1();
			num4=p.getnum2();
		}
};
int main()
{
	Item I1;
	Product P1;
	P1.setData(10,100);
	I1=P1;
	I1.showData();
}
```

//One class to another class conversion using Operator Function (overloading function)

```
#include<iostream>
using namespace std;
class Product
{
	int num1,num2;
	public:
		void setData(int x,int y)
		{
			num1=x;
			num2=y;
		}
		int getnum1()
		{
			return num1;
		}
		int getnum2()
		{
```

```
                return num2;
        }

};
class Item
{
    int num3,num4;
    public:
        Item()
        {
        }
    void operator =(Product p) //Overloading operator =
    {
        num3=p.getnum1();
        num4=p.getnum2();
    }
        void showData()
        {
            cout<<"First Value="<<num3<<"Second Value="<<num4;
        }

};
int main()
{
    Item I1;
    Product P1;
    P1.setData(10,100);
    I1=P1;
    I1.showData();
}
```

// WAP to convert an object of Dollar class to object of Rupees class.

//Another Example

By using constructors

Here, source class should have constructor. And destination class have a conversion routine. Conversion routine (constructor) would have source class object as an argument.

Example: //**conversion from polar to rectangular coordinate**

```
#include<iostream>
#include<math.h>
using namespace std;
class polar
{
public:
float r,th;
polar(){}
```

```
polar(int a,int b)
{
r=a;
th=b;
}
void show()
{
cout<<"In polar form:\nr="<<r<<" and theta="<<th;

}
};
class rectangular
{
float x,y;
public:
rectangular(){}
rectangular(polar p)
{
x=p.r*cos(p.th);
y=p.r*sin(p.th);
}
void show()
{
cout<<"\nIn Rectangular form:\nx="<<x<<"and y="<<y;

}
};
int main()
{
        polar p(5.5,3.14/2);
       p.show();
        rectangular r;
        r=p;
       r.show();

}
```

// **by using casting operator (i.e. operator function)**

Here source class contains conversion routine(i.e. operator function) and destination class have constructor.

```
// conversion from polar to rectangular coordinate
include<iostream.h>

#include<math.h>

const double pi=3.141592654;

class rectangular
```

```
{
double x,y;
public:
rectangular()
{
x=0;
y=0;
}
rectangular(double a,double b)
{
x=a;
y=b;
}

void output()
{
cout<<"("<<x<<","<<y<<")";
}
};
class polar
{
double theta,r;
public:
polar ()
{
```

```
theta=0;
r=0;
}

operator rectangular()
{
double x,y;
//float atheta=theta*pi/180;
x=r*cos(theta);
y=r*sin(theta);
return rectangular(x,y);
}

void output()
{
cout<<"\nr="<<r;
cout<<"\ntheta="<<theta;
}
};
int main()
{
rectangular r1,r2;

polar p1(3,45);
r1=p;// polar to rectangular conversion
r1.display();
}
```

Example:

```
/* Program to convert class Time to another class Minute.

#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
class Time
{
     int hrs,min;
     public:
                Time(int h,int m)
                {
                        hrs=h;
                        min=m;
                }
                Time()
                {
                        cout<<"\n Time's Object Created";
                }
                int getMinutes()
                {
                        int tot_min = ( hrs * 60 ) + min ;
                  return tot_min;
                }
                void display()
                {
                        cout<<"Hours: "<<hrs<<endl ;
                  cout<<" Minutes : "<<min <<endl ;
                }
};
class Minute
{
     int min;
     public:

       Minute()
       {
          min = 0;
       }
     void operator=(Time T)
     {
                min=T.getMinutes();
     }
     void display()
     {
                cout<<"\n Total Minutes : " <<min<<endl;
     }
};
void main()
{
```

```
    clrscr();
    Time t1(2,30);
     t1.display();
    Minute m1;
     m1.display();

    m1 = t1;    // conversion from Time to Minute

    t1.display();
    m1.display();
    getch();
}
```

Polymorphic Variable (Assignment Polymorphism):

The variable that can hold different types of values during the course of execution is called polymorphic variable. It can also be defined as the variable that is declared as one class and can hold values from subclass. In C++, it works with pointers and references. **Pure polymorphism** means a function with polymorphic variable as arguments (or parameters).

This Pointer

If only one copy of each member function exists and is used by multiple objects, then for proper data members accessing and updating, Compiler supplies an implicit pointer along with the functions names as 'this'. **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. Friend functions do not have **this** pointer, because friends are not members of a class. Only member functions have **this** pointer.

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

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

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
      // The 'this' pointer is used to retrieve the object's x
      // hidden by the local variable 'x'
      this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
     return 0;
  }
```

2) To return reference to the calling object

```
#include<iostream>
using namespace std;
 class Test
{
private:
nt x;
int y;
public:
Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
est &setX(int a) { x = a; return *this; }
Test &setY(int b) { y = b; return *this; }
void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Method overriding

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

```
#include <iostream>

using namespace std;

class BaseClass {

public:

 void disp()
{
     cout<<"Function of Parent Class";
}

};


class DerivedClass: public BaseClass{

public:

void disp() // it overrides the method of class Baseclasss
```

```
{
    cout<<"Function of Child Class";
  }
};
```