Polymorphism: Polymorphism means having many forms. That means, the same entity (object, function) behaves differently in different scenarios.

Polymorphism is a feature of OOP that allows the object or a function to perform in different ways depending on how the object or function is used. It means that, a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Let us consider a real-life example of polymorphism. A person may behave as a student at a class-room, a son at home, and a player at playground. Here, the same person shows different behaviors in different situations. This is an example of a polymorphism.

Benefits/Advantages of polymorphism:

- Polymorphism allows us to reuse the codes, by creating functions and classes that can be operated for multiple use cases.
- It allows the same variable to store multiple data-types.
- It allows the same operators to be used for different scenarios.

Types of polymorphism: Polymorphism can be categorized into two types:
(a) Compile-time polymorphism and (b) Run-time polymorphism

(a) Compile-time polymorphism: It is a type of polymorphism in which an object is bound (destined) to its function at compile-time. In this type, the compiler decides the appropriate function to execute (the called function is chosen) for a particular function call at compile-time. The binding is done based on the type of object or a pointer.

The compile-time polymorphism is also known as early binding or static binding or **static polymorphism**. It can be achieved in two ways: function overloading and operator overloading.

(b) Run-time polymorphism: It is a type of polymorphism in which an object is bound (destined) to its function at run-time. In this type, the compiler decides the appropriate function to execute (the called function is chosen) for a particular function call dynamically at run-time. The binding is done based on the location pointed to by the pointer and not according to the type of pointer.

The run-time polymorphism is also known as late binding, dynamic binding or **dynamic polymorphism**. It is achieved in two ways: function overloading and function overriding. It can be achieved with the help of function overriding with virtual functions.

Pointer to derived class: Pointers to object (object pointers) of base class are type-compatible with pointer to objects of the derived class. That means a pointer of the base class type can be used to point to the objects of base class as well as to its derived classes.

For example, if 'Parent' is a base class and 'Child' is the derived class from 'Parent', then a pointer declared as a pointer to 'Parent' can also be a pointer to 'Child'. It can be demonstrated by the following program example: *[Example program- 44:]*

Example program- 44:

```
class Parent{
  public:
  void display(){
   cout<<"From parent class"<<endl;
  }
};
class Child: public Parent{
  public:
  void display(){
    cout<<"From child class"<<endl;
  }
};

int main()
{
   Parent *ptr; //Object pointer of Parent class
   Parent p1;   //object of parent class
   Child c1;    //object of child class
   ptr =&c1;     //Object pointer of parent class pointing to child object
   ptr->display(); // It calls the function of parent class
   ((Child*) ptr)->display(); // Typecasting done. It calls the function of child class
    return 0;
}
```

Output: From parent class
From child class

In above program, the pointer 'ptr' of parent class is used to point to the child object 'c1'. It is valid with C++ because c1 is an object of class Child which is derived from the class Parent.

Note: Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. Only those members which are inherited from base class can be accessed and not the members that originally belong to the derived class. Another pointer that is declared as pointer to derived type can be used for this purpose.

Virtual function: It is a function that makes sure that the correct function is called for an object, regardless of the type of reference used for the function call.

Virtual function is declared by using a keyword 'virtual' before the normal declaration of a function. When a function is made virtual, the compiler decides which function to execute at run-time based on the type of object pointed by the pointer rather than the type of pointer. It tells the compiler to perform dynamic binding on the function, and hence achieve run-time polymorphism.

When base class pointer contains the address of the derived class object, it always executes the base class function. In this case, the compiler ignores the contents of the (base) pointer and chooses the member function that matches the type of the pointer. This issue can only be resolved by using the 'virtual' function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer. In this way runtime polymorphism is achieved.

Let us consider a following program example *[Example program- 45]*

Example program- 45:

```
class A{
  public:
    void fun(){ // no virtual function
     cout<<"Fun from parent class A."<<endl;
    }
};
class B: public A{
  public:
    void fun(){
     cout<<"Fun from child class B."<<endl;
    }
};

int main()
{
  A *ptr; //object pointer of type base class A.
  B obj; //creating object of class B.
  ptr=&obj;
  ptr->fun();
  return 0;
}
```

Output: Fun from parent class A.

Explanation of the program: In the above program, an object pointer 'ptr' is created of type A which is a base class. It is used to point to the object 'obj' of derived class B. When this pointer calls the fun( ) function, the function of the base class is executed instead of the derived class due to early binding.

The solution to this problem is declaring the fun( ) function of base class as virtual. In the above program, if we make the fun( ) a virtual function as follows:

```
class A{
  public:
    virtual void fun(){ // virtual function
     cout<<"Fun from parent class A."<<endl;
    }
};
```

Then, the output of the program would be: 'Fun from child class B'. This is due to late binding and it is the case of a run-time or dynamic polymorphism.

Example program- 46: *Another example program that demonstrates the use of virtual function/ run-time polymorphism:*

```
class Car{
 public:
    virtual void model(){
     cout<<"Model: General Car"<<endl;
    }
};
class SuperCar: public Car{
 public:
    void model(){
     cout<<"Model: Super Car"<<endl;
    }
};
class WonderCar: public Car{
 public:
    void model(){
    cout<<"Model: Wonder Car"<<endl;
   }
};
void getDetails(Car *c){
  cout<<"Type: Vehicle"<<endl;
  c->model();
}

int main()
{
   Car *ptr1, *ptr2, *ptr3;//Creating three pointers of type 'Car'.
   Car obj1;                //creating object of type 'Car'
   SuperCar obj2;       //creating object of type 'SuperCar'
   WonderCar obj3;    //creating object of type 'WonderCar'
   ptr1=&obj1;     // pointing to object 'obj1' of type 'Car'
   ptr2=&obj2;      // pointing to object 'obj2' of type 'SuperCar'
   ptr3=&obj3;       // pointing to object 'obj3' of type 'WonderCar'
   getDetails(ptr1);    //sending 'ptr1' as argument
   getDetails(ptr2);    //sending 'ptr2' as argument
   getDetails(ptr3);    //sending 'ptr3' as argument
   return 0;
}
```

Output:

```
Type: Vehicle
Model: General Car
Type: Vehicle
Model: Super Car
Type: Vehicle
Model: Wonder Car
```

Pure virtual function: A do-nothing function that is only declared but doesn't have any definition is known as pure virtual function. It is achieved by appending '= 0' at the end of the declaration of a virtual function.

This function is declared in a base class and serves as a placeholder. Any child classes derived from it have to define the function or re-declare it as a pure virtual function.

For example:

```
class Book {
    public:
    virtual void display( )=0;   //pure virtual function
};
```

In the above class Book, the display( ) function is declared as a pure virtual function. This function is defined in the child classes derived from the class Book.

Abstract class: A class that consists of at least one pure virtual function is known as abstract class. Generally, an abstract class is designed to act as a base class.

An instance (object) of such class cannot be created. But, a pointer to an abstract class can be created, and this pointer can be used for selecting the proper virtual functions.

For example:

```
class Book {
    public:
    virtual void display( )=0;   //pure virtual function
};
```

Here, Book has a pure virtual function. So, it is an abstract class. Any derived classes of it must override the function display( ) or they should declare it again as a pure virtual function. The function can be implemented in further inherited classes.

Q. Why is there the need of abstract class? (Uses of abstract class)
Solution: An abstract class is designed to act as a base class. Sometimes, implementation of all the functions cannot be provided in a base class. But these functions could be implemented in the child classes that are derived from the base class. For example, let 'Shape' be a base class and 'Circle' & 'Square' be its child classes. The Shape class has a function called draw( ). This function is not implemented in it but is implemented in its derived classes Circle and Square. So, in this type of situations, concept of abstract class is used.

An example program demonstrating the use of abstract class is given below *[Example program- 47:]*

Example program- 47:

```
class Shape{     //Abstract class
 public:
   virtual void draw()=0;
};
class Circle: public Shape{
 public:
    void draw(){
     cout<<"Drawing circle."<<endl;
    }
};
class Square: public Shape{
 public:
    void draw(){
     cout<<"Drawing square."<<endl;
    }
};
int main()
{
  Shape *ptr;
  Circle c1;
  Square s1;
  ptr=&c1;
  ptr->draw();
  ptr=&s1;
  ptr->draw();
  return 0;
}
```

Output: Drawing circle.
Drawing square.

Difference between abstract class and concrete class:

Abstract Class	Concrete Class
1. A class that has at least one pure virtual function is known as abstract class.	1. An ordinary class that has no pure virtual function is known as concrete class.
2. An object (instance) of abstract class cannot be created.	2. An object (instance) of concrete class can be created.
3. It can have unimplemented methods.	3. All the methods have to be implemented.
4. In inheritance, it is typically a base class from which other classes are derived.	4. In inheritance, it is typically a derived class that implements all the missing functionalities.

5. Example:
class Shape {
public:
virtual void draw( )=0;
}
	5. Example:
	class Circle : public Shape {
	public:
	void draw( ){
	cout<<"Drawing circle.";
	}
	};

Virtual Destructors: It is a method which ensures that both the destructors of the base class and the derived class are called at runtime to prevent any unwanted memory leakage.

Suppose an object of derived class is created dynamically using the pointer of type base class.
For example: Base *ptr = new Derived;
where, 'Base' is a base class, and 'Derived' is a derived class.

When ptr is deleted, only the destructor of the base class is called, and the destructor of derived class is not called. This may lead to memory leaks in the program.

The solution for this is to use 'virtual destructor' in the base class by using a keyword 'virtual' before destructor definition.

Eg. virtual ~ Derived( ) {

}

Now, when ptr is deleted, both the destructors of derived class and the base classes are called to release the used up memory resources.

An example program that demonstrates the use of virtual destructor is given below. *[Example program- 48]*

Example program- 48:

```
class A{
  public:
   ~A(){
     cout<<"Destructor from A"<<endl;
   }
};
class B: public A{
  public:
    ~B(){
      cout<<"Destructor from B"<<endl;
    }
};
int main()
{
   A* ptr;  //object pointer of type class 'A'
   ptr=new B;  //object of type class 'B' created dynamically
   delete ptr;  //releasing memory pointed by 'ptr'
   return 0;
}
```

Output: Destructor from A

Now in the above program, when the destructor of base class is made virtual, then both the destructors of class A and class B are called as shown below.

```
class A{
  public:
   virtual ~A(){    // Virtual destructor
     cout<<"Destructor from A"<<endl;
   }
};
class B: public A{
  public:
    ~B(){
      cout<<"Destructor from B"<<endl;
    }
};
int main()
{
   A* ptr;  //object pointer of type class 'A'
   ptr=new B;  //object of type class 'B' created dynamically
   delete ptr;  //releasing memory pointed by 'ptr'
   return 0;
}
```

Output: Destructor from B
Destructor from A

Operator Overloading: The mechanism of adding special meaning to an operator is known as operator overloading. It provides a flexibility for the creation of new definitions for most C++ operators.

Using operator overloading we can give additional meaning to normal C++ operations such as (+,-,=,<=,+= etc.) when they are applied to user defined data types.

Benefits/Use of Operator overloading: Generally, operations using operators can be performed only on basic data type.
Example: int a, b, c;
C = a + b;

But if we declare a class Complex { }; and create objects from it as:
Complex c1, c2, c3;
Then, c3 = c1 + c2; is not possible because object is user defined data type.

Operator overloading helps to define usage of operator for user defined data types, i.e. objects. That means, c3 = c1+ c2 for objects is possible if the operator (+) is overloaded.

Operators that cannot be overloaded: All operators cannot be overloaded. The operators that are not overloaded are:

- Class member access operators (. and .*)
- Scope resolution Operator (::)
- Sizeof operator(sizeof)
- Conditional Operator(? :)

General form of operator function:

```
return_type operator op(arglist) {
        function body //task defined
}
        OR
return_type classname::operator op(arglist) {
        function body   //task defined
   }
```

Where,

- return_type is the type of value returned by the specified operation.
- 'op' is the operator being overloaded.
- 'operator op' is function name, where operator is keyword.

Types of operators that are overloaded:

a) Unary operators: These are the operators that operate on single operand. Some of unary operators are: Increment operator (++), Decrement operator (--), Unary minus operator(-)
For example: ++a ; Here a is only one operand.

b) Binary operators: The operator which operates on two operands is known as binary operators. Arithematic operators (-, +, *, /), Comparision operators (>, <, < =, > =, = =), Assignment operators (+=, - =, =).
For example c=a+b where a and b are two operands.

- ❖ Operator function must be either member function (non-static) or friend function.

- ❖ Member function has no arguments for unary operators and only one for binary operators.
  Eg. Abc operator + ( ) //unary
  Abc operator + (Abc) //binary

- ❖ Friend function will have only one argument for unary operators and two for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore available for member function. This is not the case with friend functions.
  Eg. friend Abc operator + (Abc) //unary
  friend Abc operator - (Abc, Abc) //binary

- ❖ Arguments may be passed either by value or by reference. Example of prototype where the arguments are passed by reference:
  Eg. Abc operator + (Abc &a) //argument passed by reference

Mechanism for Operator overloading: Generally, the process of overloading involves the following steps:

- ➢ Create a class that defines the data type that is used in the overloading function.
- ➢ Declare the operator function operator op( ) in the public part of class. It can either be normal member function or a friend function.
- ➢ Define operator function to implement the required operations.
- ➢ Calling (invoking) the operator overloaded function.

Operator overloaded function can be invoked using expression such as:

	In case of member function	In case of friend function
For unary operators	op x or x op (eg.++x, or x++) *x.operator op();*	op x or x op (eg.++x, or x++) *operator op(x)*
For binary operators	x op y eg x+y *x.operator op(y);*	x op y eg x+y *operator op(x,y)*

- Here, op represents the operator being overloaded.
- x and y represents the objects.

A. Overloading unary operators:

1. Overloading unary (-) operator:

Example program- 49:

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

class A{

 int x;
 int y;

 public:
 A(int a, int b){
   x=a;
   y=b;
 }

 void display(){
  cout<<"Value of x: "<<x<<endl;
  cout<<"Value of y: "<<y<<endl;
 }
 void operator-(){     // - operator overloading
   x=-x;
   y=-y;
 }

};
int main()
{
  A object(2,5);
  object.display();
  -object;      //it calls the operator overloading function
  object.display();

  return 0;
}
```

Output: Value of x: 2
Value of y: 5
Value of x: -2
Value of y: -5

 # Overloading unary (–) operator using friend function:

Example program- 50:

```
class A{

 int x;
 int y;

 public:
 A(int a, int b){
   x=a;
   y=b;
 }

 void display(){
  cout<<"Value of x: "<<x<<endl;
  cout<<"Value of y: "<<y<<endl;
 }

 friend A operator-(A a);    // - operator overloading
                             // using friend function
};
A operator-(A a){      // - operator overloading
   A temp(2,5);
   temp.x=-a.x;
   temp.y=-a.y;
   return temp;  //returns object of type class A
 }

int main()
{
  A obj1(2,5);
  obj1.display();
  A obj2=-obj1;       //it calls the operator overloading function
                      //and saves the returned object in 'obj2'.
  obj2.display();

  return 0;
}
```

Output: Value of x: 2
Value of y: 5
Value of x: -2
Value of y: -5

2. Overloading the prefix (++) operator

Example program- 51:

```
class A{
  int count;
   public:
    A(){    //Default constructor
    }
    A(int a){   //Parameterized constructor
     count=a;
    }
    void display(){
     cout<<"Value of count: "<<count<<endl;
    }
   A operator ++(){   //Operator function
     A temp;
     temp.count=++count;
     return temp;
    }
};
int main()
{
  A a1(4),b1;      //Creating two objects
  a1.display();
  b1=++a1;       //Operator overloading
  a1.display();
  b1.display();
  return 0;
}
```

Output: Value of count: 4
Value of count: 5
Value of count: 5

3. Overloading the postfix (++) operator

Example program- 52:

```
class A{
  int count;
   public:
    A(){    //Default constructor
    }
    A(int a){   //Parameterized constructor
     count=a;
    }
    void display(){
     cout<<"Value of count: "<<count<<endl;
    }
   A operator ++(int){   //Operator function
     A temp;
     temp.count=count++;
     return temp;
    }
};
int main()
{
  A a1(4),b1;      //Creating two objects
  a1.display();
  b1=a1++;       //Operator overloading
  a1.display();
  b1.display();
  return 0;
}
```

Output: Value of count: 4
Value of count: 5
Value of count: 4

Note: For the overloading of postfix ++ operator, 'int' is placed in the parentheses ( ) of the operator function, so that the compiler doesn't get confused between the postfix and the prefix operator.

B. Overloading binary operators: The binary overloaded function takes the first object as an implicit operand and the second object must be passed explicitly. The data members of the first object are accessed without using the

dot operator whereas; the data members of the second object are accessed using the dot operator (if the argument is the object).
For example, if the 'op' operator is overloaded and the operator function is invoked as:

object1 op object2; then only object2 is passed explicitly as an argument

1. Overloading binary (+) operator:

Example program- 53:

```
class Distance{
  int meter, centimeter;
   public:
    Distance(){    //Default constructor
    }
    Distance(int m, int cm){   //Parameterized constructor
     meter=m;
     centimeter=cm;
    }
    void display(){
     cout<<"Distance: "<<meter<<" meter and "
      <<centimeter<<" centimeter"<<endl;
    }
    Distance operator +(Distance d){   //Operator function
     Distance temp;
     temp.meter=meter+d.meter;
     temp.centimeter=centimeter+d.centimeter;
     if(temp.centimeter >=100){
        temp.meter++;
        temp.centimeter-=100;
     }
     return temp;
    }
};

int main()
{
  Distance d1(3,60),d2(2,70),d3; //Creating three objects;
  d1.display();
  d2.display();
  d3=d1+d2;        // Binary Operator overloading
  d3.display();
  return 0;
}
```

Output:
Distance: 3 meter and 60 centimeter
Distance: 2 meter and 70 centimeter
Distance: 6 meter and 30 centimeter

 # Overloading binary (+) operator using friend function:

Example program- 54:

```
class Complex{
  int real,imag;
   public:
    Complex(){    //Default constructor
    }
    Complex(int a, int b){   //Parameterized constructor
     real=a;
     imag=b;
    }
    friend Complex operator + (Complex obj1, Complex obj2);

    void display(){
     cout<<"The number: "<<real<<" + "<<imag<<"j"<<endl;
    }
};
 Complex operator + (Complex obj1, Complex obj2){
   Complex temp;
   temp.real=obj1.real+obj2.real;
   temp.imag=obj1.imag+obj2.imag;
   return temp;
 }


 int main()
 {
   Complex c1(3,4), c2(4,1);
   Complex c3=c1+c2;    //Operator overloading
   c3.display();
   return 0;
 }
```

Output: The number: 7 + 5j

2. Overloading '+=' operator:

Example program- 55:

```
class A{
  int x,y;
   public:
    A (){   //Default constructor
    }
    A(int a, int b){  //Parameterized constructor
     x=a;
     y=b;
    }
    void display(){
     cout<<"Value x: "<<x<<endl;
     cout<<"Value y: "<<y<<endl;
    }
    A operator +=(A obj){  //Operator function
      x+=obj.x;
      y+=obj.y;
    }
};

int main()
{
  A a1(3,4),a2(2,5); //Creating two objects;
  a1.display();
  a2.display();
  a1+=a2;        // Operator overloading
  cout<<"After addition: "<<endl;
  a1.display();
  return 0;
}
```

Output:

Value x: 3
Value y: 4
Value x: 2
Value y: 5
After addition:
Value x: 5
Value y: 9

3. Overloading '<' operator:

Example program- 56:

```
class Maximum{
  int x;
   public:
    Maximum(){    //Default constructor
    }
    Maximum(int a){   //Parameterized constructor
     x=a;
    }
    Maximum operator >(Maximum obj){   //Operator function
      if(x>obj.x){
        cout<<"Maximum number is: "<<x<<endl;
      }
      else{
        cout<<"Maximum number is: "<<obj.x<<endl;
      }
    }
};
int main()
{
  Maximum m1(3),m2(5); //Creating two objects;
  m1>m2;         // Operator overloading
  return 0;
}
```

Output: Maximum number is: 5

4. Overloading '==' operator:

Example program- 57:

```
class Equal{
  int x,y;
   public:
    Equal(){    //Default constructor
    }
    Equal(int a, int b){   //Parameterized constructor
     x=a;
     y=b;
    }
    Equal operator ==(Equal obj){   //Operator function
      if(x==obj.x && y==obj.y){
        cout<<"They are equal."<<endl;
      }
      else{
        cout<<"They are not equal."<<endl;
      }
    }
int main()
{
  Equal e1(2,5), e2(2,5); //Creating two objects;
  e1==e2;         // Operator overloading
  return 0;
}
```

Output: They are equal.

Type Conversion: It is the process of converting one type of data to another type.

- Compiler automatically converts basic to another basic data type (for eg. int to float, float to int etc.) by applying type conversion rule provided by the compiler.
- The type of data to the right of the assignment operator (=) is automatically converted to the type of variable on the left.

Example: int a;
float b = 2.157;
a= b;

It converts "b" to an integer before its value is assigned to "a". So the fractional part is truncated.
However, compiler does not support automatic type conversion for user-defined data type. For example, if length1 and length2 are the objects of two different classes, then,

length1=length2 is not possible.

Similarly, user defined data type to primitive data types and vice-versa is not possible.
Eg. If length is an object of a class, and meter is an integer data, then

meter = length; // not possible.
And length =meter; //not possible

In order to realize such type of conversions, we must design conversion routines. There are three possible type conversions:

A. Conversion from basic type to class type
B. Conversion from class type to basic type
C. Conversion from one class type to another class type

A. Conversion from basic type to user-defined type/class type: To convert the data from a basic type to a user-defined type, the conversion function must be defined in the class in the form of constructor. The constructor function takes a single argument of basic data type.

General format:

```
Constructor (Basic type)
{
//converting statements
}
```

An example program that demonstrates the conversion from basic type to class type is given below. *[Example program: 4.10]*

This program converts the length 'meter' into 'feet' and 'inches'. The length 'meter' is used a basic primitive type, and the lengths 'feet' and 'inches' are used as the data members of the user-defined class type. This program uses the formula as: 1 meter= 3.3 feet, 1 foot = 12 inches.

Example program- 58:

```
class Length{
  int feet;
  float inch;
  public:
  Length(){

  }
  Length(float m){ // Type conversion using constructor //
    float f=3.3*m;
    feet=int(f);
    inch=(f-feet)*12;
  }
  void display(){
   cout<<"After conversion, length is: "<<feet<<" feet and ";
   cout<<inch<<" inches."<<endl;
  }
};
int main()
{
  Length l1;
  float meter;
  cout<<"Enter the length in meters: "<<endl;
  cin>>meter;
  l1=meter; // float to user-defined data type conversion //
  l1.display();
  return 0;
}
```

Output: Enter the length in meters:
5.5 (entered by user)
After conversion, length is: 18 feet and 1.8 inches.

B. User-defined type/class type to basic type: To convert the data from a user-defined type to a basic type, the conversion function must be defined in the class in the form of the casting operator function. The casting operator function is defined as an overloaded basic data types which takes no arguments. It converts the data members of an object to basic data types and returns a basic data item.

General format:
Operator basic_data ( ) {
//conversion statements
}

An example program that demonstrates the conversion from class type to basic type is given below. *[Example program 4.11]* This program converts the length 'feet' and 'inches' to meter. The lengths 'feet' and 'inches' are used as the data members of the user-defined class type and the length 'meter' is used a basic primitive type.

Example program- 59:

```
class Length{
  int feet;
  float inch;
  public:
  void getData(){
    cout<<"Enter the length: "<<endl;
    cout<<"Feet: ";
    cin>>feet;
    cout<<"inch: ";
    cin>>inch;
  }
  operator float(){ //type conversion using casting operator
   float f=feet + (inch/12);
   float m= f/3.3;
   return m;
  }
};
int main()
{
  Length l1;
  l1.getData();
  float meter;
  meter=l1; //user-defined to float data type conversion //
  cout<<"After conversion, the length is: "<<meter<<" meters";
  return 0;
}
```

Output: Enter the length:
Feet: 18 (entered by user)
inch: 1.8 (entered by user)
After conversion, the length is: 5.5 meters

Example program- 60: *Write a program to convert centigrade into Fahrenheit temp using type conversion routine. Use formula: F = (C * 1.8) + 32.*
(If the question asks just opposite, the formula to convert Celsius to Fahrenheit is: C = (F - 32) /1.8).

Note: Here, the question has not mentioned anything about which conversion routine to follow, so you can use any conversion routine you prefer.

Here, we are going to use primitive to user-defined data type conversion routine.

```
class Temperature{
  float fahrenheit;
  public:
  Temperature(){

  }
  Temperature(float c){ //Type conversion using constructor//
   fahrenheit = (c * 1.8) + 32;
  }
  void display(){
   cout<<"The converted temperature is: "<<fahrenheit<<" degree F"<<endl;
  }

};
int main()
{
  Temperature t;
  float celsius;
  cout<<"Enter the temperature in degree celsius: ";
  cin>>celsius;
  t=celsius; //float to user-defined data type conversion //
  t.display();
  return 0;
}
```

Output: Enter the temperature in degree celsius: 37.5 (entered by user)
The converted temperature is: 99.5 degree F

C. User-defined data type (class type) to User-defined data type (class type): When a data of one class type is converted into data of another class type, it is called conversion of one class to another class type.

For example: objA = objB;

Here, objA is an object of class A and objB is an object of class B. The data of type class B is converted to the data of type class A and converted value of objB is assigned to the objA. Since the conversion takes place from class B to class A, B is known as source class and A is known as destination class. This type of conversion is carried out by either:

1. constructor or
2. casting operator function

So, one of the two alternatives can be chosen for the data conversion. The choice depends on whether we put the conversion routine in the destination class or in the source class. If we want the conversion routine to be located in the source class then the operator function method is chosen. Or, if we want the conversion routine to be located in the destination class then the constructor method is chosen.

1. Conversion Routine in source object/Using casting operator function:

General format:

```
class Destination {
};
class Source{
  public:
    operator Destination( ) {
      // code for conversion
        return (Destination type);
    }
};
// d1 is the object of class Destination and s1 is the object of class Source
d1=s1;    // casting operator function is called
```

An example program that demonstrates the conversion routine using casting operator function is given below. *[Example program- 61]*

This program converts the feet and inch of one class type to the meter and cm of another type.

Example program- 61:

```
class Distance2{
 int meter;
 float cm;
  public:
    void getData(int m, float c){
     meter=m;
     cm=c;
    }
    void display(){
      cout<<"The distance is "<<meter<<" meter ";
      cout<<"and "<<cm<<" cm."<<endl;
    }
};
class Distance1{
  int feet;
  float inch;
  public:
    void getData(){
     cout<<"Enter the distance in feet and inch."<<endl;
     cout<<"Feet: ";
     cin>>feet;
     cout<<"Inch: ";
     cin>>inch;
    }
    operator Distance2(){ /**Type conversion using constructor**/
     float sum=feet+(inch/12);  //the total distance in feet
     float m= sum/3.3;          //converting feet to meter
     int meter=int(m);
     float cm=(m-meter)*100;
     Distance2 temp;
     temp.getData(meter,cm);
     return temp;
    }
};
int main()
{
  Distance1 d1;
  d1.getData();
  Distance2 d2;
  d2=d1; /** One class to another class data type conversion **/
  d2.display();
  return 0;
}
```

Output: Enter the distance in feet and inch.
Feet: 18 (entered by user)
Inch: 1.8 (entered by user)
The distance is 5 meter and 50 cm.

2. Conversion routine in destination object/Using constructor:

General format:

```
class Source {
};
class Destination{
  public:
    Destination(Source objectS ) {
      // code for conversion
    }
};
// d1 is the object of class Destination and s1 is the object of class Source
d1=s1;   // constructor called
```

An example program that demonstrates the conversion routine using constructor is given below. *[Example program 4.14]*

Example program- 62:

```
class Distance1{
  int feet;
  float inch;
  public:
    void getData(){
     cout<<"Enter the distance in feet and inch."<<endl;
     cout<<"Feet: ";
     cin>>feet;
     cout<<"Inch: ";
     cin>>inch;
    }
    int getFeet(){
     return feet;
    }
    float getInch(){
     return inch;
    }
};
```

```
class Distance2{
 int meter;
 float cm;
  public:
    Distance2(){

    }
    Distance2(Distance1 d1){ /**Type conversion using constructor**/
      int f=d1.getFeet();  //accessing feet of d1 object.
      float i=d1.getInch();  //accessing inch of d1 object.
      float sum=f+(i/12);  //the total distance in feet
      float m= sum/3.3;   //converting feet to meter
      meter=int(m);
      cm=(m-meter)*100;
    }
    void display(){
      cout<<"The distance is "<<meter<<" meter ";
      cout<<"and "<<cm<<" cm."<<endl;
    }
};
int main()
{
  Distance1 d1;
  d1.getData();
  Distance2 d2;
  d2=d1; /** One class to another class data type conversion **/
  d2.display();
  return 0;
}
```

Output: Enter the distance in feet and inch.
Feet: 18 (entered by user)
Inch: 1.8 (entered by user)
The distance is 5 meter and 50 cm.