UNIT III POLYMORPHISM

ADT conversions:

Overloading:

Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. For example, the compiler acts differently with regards to the subtraction operator "-" depending on how the operator is being used. When it is placed on the left of a numeric value such as -48, the compiler considers the number a negative value. When used between two integral values, such as 80-712, the compiler applies the subtraction operation. When used between an integer and a double-precision number, such as 558-9.27, the compiler subtracts the left number from the right number; the operation produces a double-precision number. When the - symbol is doubled and placed on one side of a variable, such as --Variable or Variable--, the value of the variable needs to be decremented; in other words, the value 1 shall be subtracted from it. All of these operations work because the subtraction operator "-" has been reconfigured in various classes to act appropriately.

Unary Operator Overloading:

You can overload both the unary and binary forms of the following operators:

You cannot overload the preprocessor symbols `#` and `##`.

An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type.

You cannot change the precedence, grouping, or the number of operands of an operator.

An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.

You must declare the overloaded `=`, `[]`, `()`, and `->` operators as nonstatic member functions to ensure that they receive lvalues as their first operands.

You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter. Suppose a unary operator `@` is called with the statement `@t`, where `t` is an object of type `T`. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

Example:

```
#include<iostream.h>
#include<conio.h>

class complex
{
     int a,b,c;
    public:
        complex(){}
        void getvalue()
       {
                  cout<<"Enter the Two Numbers:";
                  cin>>a>>b;
       }



  void operator++()
      {
                  a=++a;
                  b=++b;
       }

       void operator--()
       {
                  a=--a;
                  b=--b;
        }

        void display()
        {
                  cout<<a<<"+\t"<<b<<"i"<<endl;
         }
};

void main()
{
     clrscr();
     complex obj;
     obj.getvalue();
     obj++;
     cout<<"Increment Complex Number\n";
     obj.display();
     obj--;
     cout<<"Decrement Complex Number\n";
     obj.display();
     getch();
}
```

Binary Operator overloading:

A binary operator can be overloaded as a non-static member function with one argument or as a global function with two arguments (one of those arguments must be either a class object or a reference to a class object).

```
#include <iostream.h>

struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display();
}
```

Function selection:

Pointer operators: