OBJECT ORIENTED PROGRAMMING (PC-CS-203A)

UNIT-III

Prepared By:

Er. Tanu Sharma

(A.P. CSE Deptt., PIET)

Syllabus (Unit-3)

Polymorphism, Pointer to Derived class, Virtual Functions, Pure Virtual Function, Abstract Base Classes, Static and Dynamic Binding, Virtual Deconstructors.

Fundamentals of Operator Overloading, Rules for Operators Overloading, Implementation of Operator Overloading Like <<, >>, Unary Operators, Binary Operators.

Pointers

- A pointer is a derived data type that refers to another data variable.
- Stores variable's memory address rather than data
- It defines where to get the value of a specific data variable instead of defining actual data.
- A pointer can also refer to another pointer.

Declaring and Initializing Pointer

- The declaration is based on data type of variable it points to.
- General form:

  data-type *pointer-variable;

- Example:

```
int  *ptr;      //pointer variable which points to integer data type.
int a;
ptr=&a;      //assigns the address of variable a to pointer ptr
```

OR

```
int  *ptr, a;
ptr=&a;
```

EXAMPLE OF USING POINTERS

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a, *ptr1, **ptr2;
clrscr();
ptr1 = &a;
ptr2=&ptr1;
cout << "The address of a : " << ptr1 << "\n";
cout << "The address of ptr1 : " << ptr2;
cout << "\n\n";
cout << "After incrementing the address values:\n\n";
ptr1+=2;
cout << "The address of a : " << ptr1 << "\n";
ptr2+=2;
cout << "The address of ptr1 : " << ptr2 << "\n";
}
```

Output of Program

```
The address of a :      0x8fb6fff4
The address of ptr1:    0x8fb6fff2
After incrementing the address values:
The address of a :      0x8fb6fff8
The address of a :      0x8fb6fff6
```

Manipulation of pointers

- The content of pointer variable can be directly accessed using **Dereference operator**
- E.g. *pointer_variable
- It can also change the content of memory location.
- A pointer must be initialized before dereferencing a pointer, otherwise it will cause runtime error.

MANIPULATION OF POINTERS

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a=10, *ptr;
ptr = &a;
clrscr();
cout << "The value of a is : " << a;
cout << "\n\n";
*ptr=(*ptr)/2;
cout << "The value of a is : " << (*ptr);
cout << "\n\n";
}
```

Output of Program

```
The value of a is : 10

The value of a is : 5
```

Pointer Expression and Arithmetic

- Following arithmetic operations can be performed on pointers:
  - Can be incremented (++) or decremented (--)
  - Any integer can be added to or subtracted from a pointer
  - One pointer can be subtracted from another.
  - E.g.

    int a[6];

    int *aptr;

    aptr=&a[0];
  - The pointer variable, **aptr** refers to base address of variable **a**.
  - We can increment or decrement it as:
    - **aptr++ or ++aptr**
    - **aptr-- or --aptr**

ARITHMETIC OPERATIONS ON POINTERS

```
#include<iostream.h>
#include<conio.h>
void main()
{
     int num[]={56,75,22,18,90};
     int *ptr;
     int i;
     clrscr();
     cout << "The array values are:\n";
     for(i=0;i<5;i++)
               cout<< num[i]<<"\n";
     /* Initializing the base address of str to ptr */
     ptr = num;
     /* Printing the value in the array */
     cout << "\nValue of ptr    : "<< *ptr;
     cout << "\n";
     ptr++;
     cout<<"\nValue of ptr++    : "<<*ptr;
     cout << "\n";
     ptr--;
     cout<<"\nValue of ptr--    : "<<*ptr;
     cout << "\n";
     ptr=ptr+2;
```

```
    cout<<"\nValue of ptr+2   : "<<*ptr;
    cout << "\n";
    ptr=ptr-1;
    cout <<"\nValue of ptr-1: "<< *ptr;
    cout << "\n";
    ptr+=3;
    cout<<"\nValue of ptr+=3: "<<*ptr;
    ptr-=2;
    cout << "\n";
    cout<<"\nValue of ptr-=2: "<<*ptr;
    cout << " \n";
    getch();
}
```

Output of Program 9.3:

```
The array values are:
56
75
22
18
90
Value of ptr     :  56
Value of ptr++   :  75
Value of ptr--   :  56
Value of ptr+2   :  22
Value of ptr-1   :  75
Value of ptr+=3  :  90
Value of ptr-=2  :  22
```

Using Pointers with Arrays

ARRAYS	POINTERS
• Arrays refers to a block of memory space. • Memory address of array cannot be changed.	• Pointers do not refer to any section of memory. • Content of pointers variables, such as memory addresses that it refers to, can be changed. **int *nptr;** **nptr = number[0];** **OR** **nptr = number;** **// nptr points to first element of integer array.** **// number and &number[0] can be used interchangeably.**

```
#include <iostream.h>

void main()
{
        int numbers[50], *ptr;
        int n,i;
        cout << "\nEnter the count\n";
        cin >> n;
       cout << "\nEnter the numbers one by one\n";
       for(i=0;i<n;i++)
       cin >> numbers[i];
       /* Assigning the base address of numbers to ptr and initializing
       the sum to 0*/
       ptr = numbers;
       int sum=0;
      /* Check out for even inputs and sum up them*/
      for(i=0;i<n;i++)
  {
      if (*ptr%2==0)
            sum=sum+*ptr;
      ptr++;
  }

  cout << "\n\nSum of even numbers: " << sum;
}
```

Output of Program

```
Enter the count
5
Enter the numbers one by one
10
16
23
45
34
Sum of even numbers:    60
```

Array of Pointers

- Array of pointers represents a collection of addresses
- Points to an array of data items.
- Data elements can be accessed directly or by dereferencing the elements of pointer array.
- General form:
  - int *inarray[10];

ARRAYS OF POINTERS

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
void main()
{
      int i=0;
      char *ptr[10] = {
            "books",
            "television",
            "computer",
            "sports"
            };
      char str[25];
      clrscr();
      cout << "\n\n\n\nEnter your favorite leisure pursuit: " ;
      cin >> str;
```

```
  for(i=0; i<4; i++)
  {
          if(!strcmp(str, *ptr[i]))
          {
          cout << "\n\nYour favorite pursuit " << " is available here"
          << endl;
          break;
          }
   }
    if(i==4)
       cout << "\n\nYour favorite " << " not available here" << endl;
  getch();
}
```

Output

```
Enter your favorite leisure pursuit: books

Your favorite pursuit is available here
```

Pointer to Functions

- Also known as callback function.
- Function is passed as a pointer.
- Function pointers cannot be dereferenced.
- General form:
  - data_type (*function_name) ();
- Example:
  - int (*num_function (int x));
- Declaring a pointer only creates a pointer.
  - The task which is to be performed by the pointer must also be defined.
- The function must have same return type and arguments.

```
//POINTER TO FUNCTION

#include<iostream.h>
#include<conio.h>
void (*funptr)(int,int);
void add(int i,int j)
{
        cout<<endl<<i<<" + "<<j<<" = "<<i+j;
}
void subtract(int i,int j)
{
        cout<<endl<<i<<" - "<<j<<" = "<<i-j;
}
void main()
{
        funptr=&add;
        funptr(1,2);
        funptr=&subtract;
        funptr(5,3);
        getch();
}
```

OUTPUT:

1 + 2 = 3
5 – 3 = 2

Pointers to Objects

- Useful in creating objects at run time.
- Object pointer access the public members of an object.

- Example:

```
item x;
item *ptr=&x;     //ptr is initialized with address of x.
```

- Member functions of class item can be called in two ways:
  - By using the dot operator and the object.

```
x.getdata(100, 75.50);
x.show();
(*ptr).show();
```

  - By using the arrow operator and the object pointer

```
ptr->getdata(100, 75.50);
ptr->show();
```

POINTERS TO OBJECTS

```
#include <iostream>

using namespace std;

class item
{
    int code;
    float price;
  public:
    void getdata(int a, float b)
    {
        code = a;
        price = b;
    }

    void show(void)
    {
        cout << "Code : " << code << "\n";
        cout << "Price: " << price << "\n";
    }
};

const int size = 2;
```

```
const int size = 2;

int main()
{
    item *p = new item [size];
    item *d = p;
    int x, i;
    float y;

    for(i=0; i<size; i++)
    {
        cout << "Input code and price for item" << i+1;
        cin >> x >> y;
        p->getdata(x,y);
        p++;
    }

    for(i=0; i<size; i++)
    {
        cout << "Item:" << i+1 << "\n";
        d->show();
        d++;
    }

    return 0;
}
```

The output of Program

```
Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code : 40
Price: 500
Item:2
Code : 50
Price: 600
```

Polymorphism

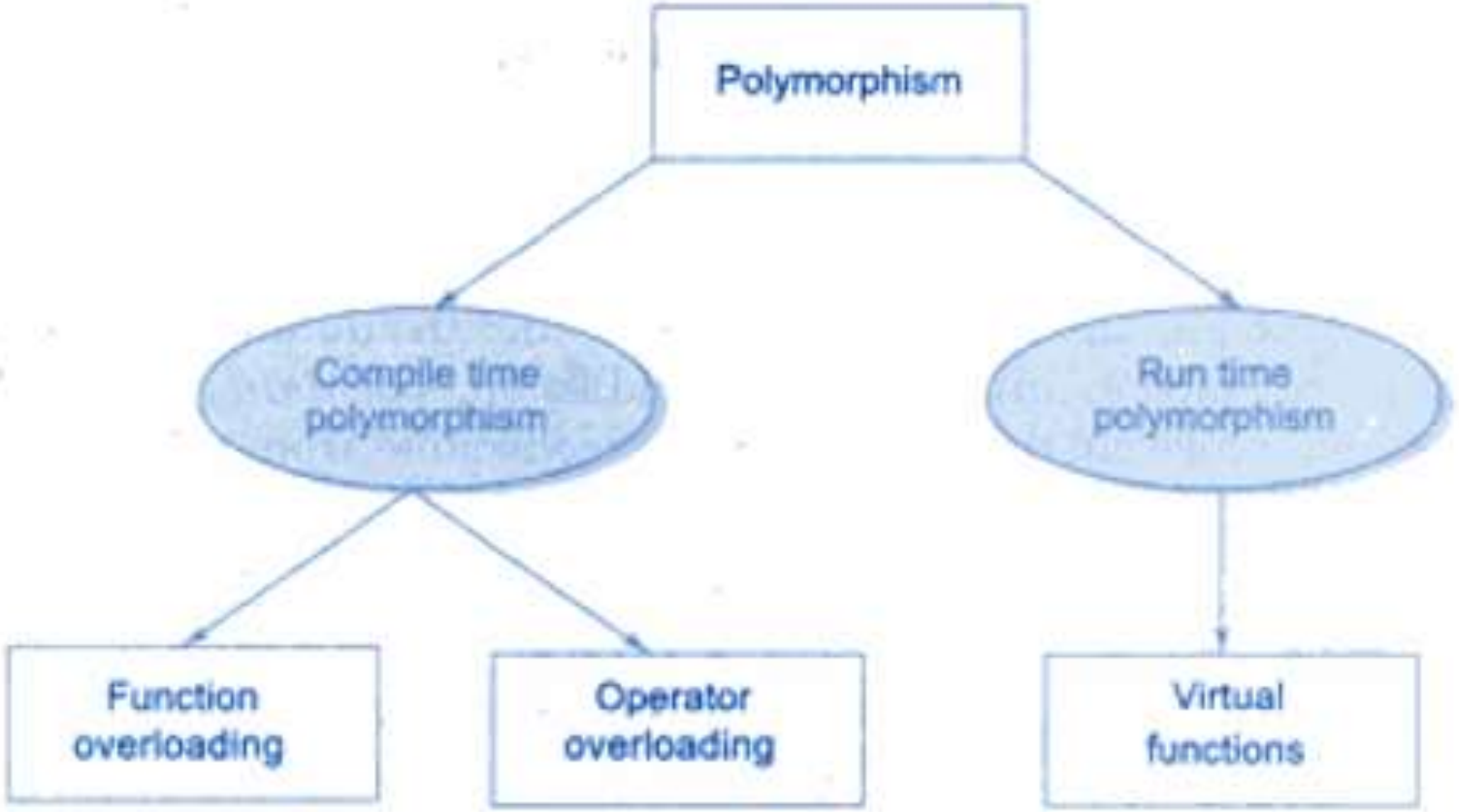

Fig. 9.1 ⇔ *Achieving polymorphism*

Compile time polymorphism

- Means an object is bound to its function call at compile time.
- There is no ambiguity at compile time about which function is linked to a particular function call.
- Known as **Early Binding** or **Static Binding**.
- Achieved in two ways:
  - Function overloading
  - Operator overloading

Need for run time polymorphism

- Used in large applications.
- Linking of function call to a particular class at run time.
- It is not known before program execution which function will be invoked till an object actually makes a function call during program's execution.
- Known as **Late Binding** or **Dynamic Binding**.
- Can be achieved with the help of virtual functions.

Pointers to derived classes

- Pointers can be used not only for base objects but also to the objects of derived class.
- A single pointer variable can be made to point to objects belonging to different classes.

```
B *cptr;        // pointer to class B type variable
B  b;           // base object
D  d;           // derived object
cptr = &b;      // cptr points to object b
```

  - We can make cptr to point to object d:

    cptr = &d; //cptr points to object d.

- Pointer of base class pointing to derived class can only access those members which are inherited from B and not the members that originally belongs to D.

- If both the base class and the derived class has a member function of same name, any reference to that member by pointer of base class will always access the member of base class.
- A pointer of derived class need to be declared to access all the members which originally belongs to derived class.

POINTERS TO DERIVED OBJECTS

```
#include <iostream>

using namespace std;

class BC
{
  public:
     int b;
     void show()
     { cout << "b = " << b << "\n";}
};
class DC : public BC

{
  public:
     int d;
     void show()
     { cout << "b = " << b << "\n"
            << "d = " << d << "\n";
     }
};
```

```
int main()
{
        BC *bptr;                       // base pointer
        BC base;
        bptr = &base;                   // base address

        bptr->b = 100;                  // access BC via base pointer
        cout << "bptr points to base object \n";
        bptr -> show();
        // derived class
        DC derived;
        bptr = &derived;                // address of derived object
        bptr -> b = 200;                // access DC via base pointer

        /* bptr -> d = 300;*/           // won't work
        cout << "bptr now points to derived object \n";
        bptr -> show();                 // bptr now points to derived object

        /* accessing d using a pointer of type derived class DC */

        DC *dptr;                       // derived type pointer
        dptr = &derived;
        dptr->d = 300;

        cout << "dptr is derived type pointer\n";
        dptr -> show();

        cout << "using ((DC *)bptr)\n";
        ((DC *)bptr) -> d = 400;
        ((DC *)bptr) -> show();

        return 0;
}
```

Program 9.11 produces the following output:

```
bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400
```

Virtual Functions

- The ability to refer to objects without regard to their classes is essential to achieve polymorphism.
- By using pointer to base class, all derived objects can also be referred.
- If in case of same name of member function in both base class and derived class, only base class function can be accessed.
- To overcome this limitation, virtual functions can be used.

- Earlier,
  - The compiler ignores the content of pointer (i.e. the class to which pointer variable points to).
  - Matches only the type of pointer.

- In virtual function,
  - Function in base class is declared as virtual.
  - It is determined only at run time, which function to call
  - It depends upon the type of object pointed to by base pointer.
  - It does not depend upon the type of pointer.

VIRTUAL FUNCTIONS

```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};
```

```
int main()
{
        Base B;
        Derived D;
        Base *bptr;

        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Derived version

        return 0;
}
```

The output of Program

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

- Run time polymorphism can only be achieved by a virtual function is accessed through a pointer of base class.

Rules for Virtual Functions

- Virtual functions must be the members of some class.
- They cannot be static.
- Can be accessed by using object pointers.
- Virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- The prototypes of all base class and derived class must be identical.
  - If there are two functions with same name and different prototypes, they are considered as overloaded functions but not virtual functions.
- There cannot be virtual constructors but virtual destructors.
- A base class pointer can access any type of derived object, but a derived class pointer cannot access an object of base of base type.
- If a virtual function is defined in the base class, it need not necessarily redefined in derived class. Then, class will invoke the base function.

Pure Virtual Functions

- A virtual function is declared inside the base class and redefined in derived classes.
- The function in base class is rarely used for performing any task.
- Such functions are called **do-nothing functions**.
- General form:

  virtual void display()=0;

- Such functions are called **Pure Virtual Functions**.

- A pure virtual function declared in base class has no definition.
- Each derived is required to define the function or re-declare it as a pure virtual function.
- A class containing pure virtual function cannot be used to declare any objects of its own.
- Such classes are called **Abstract Base Classes**.

```
#include <iostream>

using namespace std;

class Balagurusamy          //base class
{
public:
virtual void example()=0;    //Denotes pure virtual Function Definition
};
class C:public Balagurusamy       //derived class 1
{
public:
        void example()
        {
        cout<<"C text Book written by Balagurusamy";
        }
};
class oops:public Balagurusamy    //derived class 2
{
public:
        void example()
        {
        cout<<"C++ text Book written by Balagurusamy";
        }
};
void main()
{
 Balagurusamy* arra[2];

C e1;
oops e2;
arra[0]=&e1;
arra[1]=&e2;

arra[0]->example();
arra[1]->example();
}
```

Virtual Destructors

```
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

OPERATOR OVERLOADING

Operator Overloading

- The ability to provide the operators with a special meaning for a data type is called **Operator Overloading**.
- This mechanism provides a flexible option for creation of new definitions for operators used in C++.
- While overloading, only semantics of operators can be changed and not the syntax.
- All operators except the following can be overloaded:
  - Class member access operator (., .*)
  - Scope resolution operator (::)
  - Size operator (**sizeof**)
  - Conditional operator (?:)

Operator Overloading Definition

- General form:

  Return-type classname::operator op(arglist)
  {
    function body
  }

  Where,
  - **return-type**: type of value returned by the specified operation
  - **op**: operator being overloaded
  - **operator op**: function name
  - **operator**: keyword

- Operator overloading function should be declared in public section of class.

Types of Operator Functions

- Operator functions can be:
  - Member functions
  - Friend functions
- Member functions
  - For unary operators: no operands
  - For binary operators: only one operand is used
  - Object used to call member function is passed implicitly
- Friend functions
  - For unary operators: one operand is used
  - For binary operators: two operands are used

Few prototypes of operator overloading

```
vector operator+(vector);                    // vector addition
vector operator-();                          // unary minus
friend vector operator+(vector,vector);      // vector addition
friend vector operator-(vector);             // unary minus
vector operator-(vector &a);                 // subtraction
int operator==(vector);                      // comparison
friend int operator==(vector,vector)         // comparison
```

Invoking overloaded functions

- For friend function
  - For unary operators
    - ***op* x** or **x *op***
    - interpreted as **operator op (x)**
  - For binary operators
    - **x *op* y**
    - interpreted as **operator op (x, y)**

- For member function
  - For binary operators
    - **x *op* y**
    - interpreted as **x.operator op (y)**

Overloading Unary Minus

```
#include <iostream>

using namespace std;

class space
{
        int x;
        int y;
        int z;
 public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();     // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}
```

```
int main()
{
        space S;
        S.getdata(10, -20, 30);
      cout << "S : ";
      S.display();

      -S;                          // activates operator-() function

      cout << "S : ";
      S.display();

      return 0;
}
```

The Program 7.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

Using Friend Function

```
friend   void operator-(space &s);                    // declaration
void operator-(space &s)                              // definition
  {
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
  }
```

- Argument is passed by reference only
- If passed by value, no change will be reflected in the called object.

Overloading Binary Operators

```
#include <iostream>

using namespace std;

class complex
{
        float x;                                // real part
        float y;                                // imaginary part
  public:
        complex(){ }                            // constructor 1
        complex(float real, float imag)         // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};

complex complex :: operator+(complex c)
{
        complex temp;                   // temporary
        temp.x = x + c.x;               // these are
        temp.y = y + c.y;               // float additions
        return(temp);
}

void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
```

```
}

int main()
{
        complex C1, C2, C3;             // invokes constructor 1
        C1 = complex(2.5, 3.5);         // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();

        return 0;
}
```

The output of Program

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

```
#include <iostream.h>

  const size = 3;

  class vector
  {
        int v[size];
   public:
        vector();                          // constructs null vector
        vector(int *x);                    // constructs vector from array
        friend vector operator *(int a, vector b);        // friend 1
        friend vector operator *(vector b, int a);        // friend 2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
  };

  vector :: vector()
  {
        for(int i=0; i<size; i++)
              v[i] = 0;
  }

  vector :: vector(int *x)
  {
        for(int i=0; i<size; i++)
              v[i] = x[i];
  }
```

```
vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}

istream & operator >> (istream &din, vector &b)

{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
```

```
ostream & operator <<  (ostream &dout, vector &b)
  {
        dout << "(" << b.v [0];

        for(int i=1; i<size; i++)
              dout << ", " << b.v[i];
        dout << ")";
        return(dout);
  }

  int x[size] = {2,4,6};

  int main()
  {
        vector m;              // invokes constructor 1
        vector n = x;          // invokes constructor 2

        cout << "Enter elements of vector m " << "\n";
        cin >> m;              // invokes operator>>() function
```

```
    cout << "\n";
    cout << "m = " << m << "\n";          // invokes operator <<()

    vector p, q;

    p = 2 * m;                  // invokes friend 1
    q = n * 2;                  // invokes friend 2

    cout << "\n";
    cout << "p = " << p << "\n";          // invokes operator<<()
    cout << "q = " << q << "\n";

    return 0;
}
```

Shown below is the output of Program

```
Enter elements of vector m
5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

END OF UNIT-III