Experiment 4 **Date:-_____**

Aim:To study fundamentals of Operator Overloading

Theory: the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism.It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

Operator overloading is a key feature that helps in making code more intuitive and expressive when working with complex objects.Operator overloading can be done as either a member function or a non-member (friend) function.

Here's the general syntax for both:

Member Function Syntax:

```
return_type operator operator_symbol(class_name rhs_object){
        body of the function
}
```

Friend Function Syntax:

```
return_type operator operator_symbol(class_name rhs_object){
        body of the function
}
```

Advantages:

Intuitive Syntax: Operator overloading provides the ability to perform operations using familiar symbols (like +,-, etc.) even with custom objects, resulting in code that feels natural.

Code Readability: By overloading operators, complex operations on objects can be simplified, improving code clarity.

Reusability and Modularity: Operators can be overloaded in such a way that they work consistently across different contexts, making code more reusable.

PROGRAM 4[A]: Write a C++ program to understand overloading of unary prefix & postfix operators to perform increment and decrement operations on objects.

Program:

```
 #include<iostream>
using namespace std;
class test{
    int a;
    public:
    test(int p){
        a=p;
    }
    void operator - (){
        a=-a;
    }
    void operator ++(){
        a=a+1;
    }
    void operator ++(int){
        a=a+1;
    }
    void operator --(){
        a=a-1;
    }
    void operator --(int){
        a=a-1;
    }
    void display(){

cout<<"Result : "<<a<<endl;;
    }

};
int main(){
    test t1(1);
    t1++;
    t1.display();
    ++t1;
    t1.display();
    t1--;
    t1.display();
    --t1;
    t1.display();
    -t1;
    t1.display();
    return 0;
}
```

Output:

```
Result : 2
Result : 3
Result : 2
Result : 1
Result : -1
```

Program 4[B]: Write a C++ program to understand overloading of binary operators to perform the following operations on the objects of the class:

i. x = 5 + y

ii. x = x * y where x & y are objects of the class

iii. x = y - 5

Program:

```
#include<iostream>
using namespace std;
class calc{
    int a;
    public:
    calc(){
        a=0;
    }
    calc(int y){
        a=y;
    }
    friend calc operator +(int q,calc &
c22){
        calc c2;
        c2.a=q+c22.a;
        return c2;
    }
    calc operator -(int r){
        calc c3;
        c3.a=a-r;
        return c3;
  }
    calc operator *(calc s){
        calc c4;
        c4.a=a* s.a;
        return c4;
    }
    void display(){
        cout<<"Result is : "<<a<<endl;
    }
};
```

Output:

```
int main()
    {
        calc y(10),x;
        x=5+y;
        x.display();
        x= x*y;
        x.display();
        x=y-5;
        x.display();
        return 0;
    }
```

Output:

```
Result is : 15
Result is : 150
Result is : 5
```

Program 4[C]: Write a C++ program to overload binary stream insertion (<<) & extraction (>>) operators when used with objects.

Program:

```
#include<iostream>
using namespace std;
class test{
    int a;
    public:
    test(){
        a=0;
    }
    friend void operator >> (istream
&c1,test &t11){
        c1>>t11.a;
    }
    friend void operator << (ostream
&c2,test &t11){
        c2<<t11.a;
    }
};
int main(){
    test t;
    cout<<"Enter a number"<<endl;
    cin>>t;
    cout<<"The number entered was \n";
    cout<<t;
    return 0;
}
```

Output:

```
Enter a number
21
The number entered was
21
```

Program 4[D]: Write a C++ program using class string to create two strings and perform the following operations on the strings
i. To add two string type objects (s1 = s2 + s3) where s1,s2,s3 are objects
ii. To compare two string lengths to print which string is smaller & print accordingly.

Program:

```
#include<iostream>
#include<string>
using namespace std;
class String{
    string str;
    int len;
    public:
        String(){
            str="";
        }
        friend istream &operator>>(istream
&in,String &s){
            cout<<"Enter a string: ";
            in>>s.str;
            s.len=s.str.length();
            return in;
        }
        friend ostream &operator<<(ostream
&out,String &s){
            out<<"The string is: ";
            out<<s.str;
            out<<"\nThe length of the string is:
";
            out<<s.len<<endl;
            return out;
        }
        String operator+(String &s){
            String temp;
            temp.str=str+ " " +s.str;
            cout<<"The string is: "<<temp.str;
            return temp;
        }
        friend void compare(String &s2,String
&s3){
            if(s2.len>s3.len)
                cout<<"\nLength of "<<s2.str<<"
> "<<"Length of "<<s3.str;
            else if(s2.len<s3.len)
                cout<<"\nLength of "<<s3.str<<"
> "<<"Length of "<<s2.str;
            else
                cout<<"\nLength of "<<s3.str<<"
= "<<"Length of "<<s2.str;

        }
};
int main(){
    String s1,s2,s3;
    cin>>s2;
    cin>>s3;
    cout<<s2;
    cout<<s3;
    s1=s2+s3;
    compare(s2,s3);
}
```

Output:

```
Enter a string: Shikhaa
Enter a string: Prabhudesai
The string is: Shikhaa
The length of the string is: 7
The string is: Prabhudesai
The length of the string is: 11
The string is: Shikhaa Prabhudesai
Length of Prabhudesai > Length of Shikhaa
```

Program 4[E]: Write a C++ program to create a vector of ‘n’ elements (allocate the memory dynamically) and then multiply a scalar value with each element of a vector. Also show the result of addition of two vectors.

Program:

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

class Vector{
    int* a;
    int size;
    public:
    Vector(int size2=3){
        size=size2;
        a=new int[size];
        for(int i=0;i<size;i++){
            a[i]=0;
        }
    }
    void getdata(){
        cout<<"Enter the elements of the
Vector"<<endl;
        for(int i=0;i<size;i++){
            cin>>a[i];
        }
    }
    Vector operator*(int p){
        Vector v3(size);
        for(int i=0;i<size;i++){
            v3.a[i]=a[i]*p;
        }
        return v3;
    }
    Vector operator-(Vector& v){
        if(size!=v.size){
            cout<<"Vector sizes are not the
same"<<endl;
            return Vector();
        }
        Vector result(size);
        for(int i=0;i<size;i++){
            result.a[i]=a[i]-v.a[i];
        }
        return result;
    }
    void display(){
        for(int i=0;i<size;i++){
            cout<<" "<<a[i];
        }
        cout<<endl;
    }
};
int main(){
    int n,scal;
    cout<<"Enter the size of the vector: ";
    cin>>n;
    Vector v1(n),v2(n),v3(n);
    v1.getdata();
    printf("Vector 1 is : ");
    v1.display();
    cout<<"Enter scalar number: ";
    cin>>scal;
    v2=v1*scal;
    v3=v1-v2;
    cout<<"Scalar Multiplication: ";
    v2.display();
    printf("Vector 2 is : ");
    v2.display();
    cout<<"Vector Subtaction : ";
    v3.display();
}
```

Output:

Output:

```
Enter the size of the vector: 3
Enter the elements of the Vector
10
20
30
Vector 1 is :  10 20 30
Enter scalar number: 2
Scalar Multiplication:  20 40 60
Vector 2 is :  20 40 60
Vector Subtaction :  -10 -20 -30
```

Program 4[F]: Write a program for manipulating linked list supporting node operations as follows: node=node+2; node=node-3;

The first statement creates a new node with node information 2 and the second statement deletes a node with node information 3. overload the new and delete operators.

Program:

```
#include<iostream>
using namespace std;
class Node{
    int data;
    Node *next;
    public:
        Node(int v){
            data=v;
            next=NULL;
        }
        void *operator new(size_t size){
            void *node=::operator new(size);
            return node;
        }
        void operator delete(void *node){
            ::operator delete(node);
        }
        friend class linkedList;
};
class linkedList{
    public:
        Node *head;
        linkedList(){
            head=NULL;
        }
        linkedList &operator +(int val){
            Node *temp=new Node(val);
            temp->data=val;
            temp->next=NULL;
            if(head==NULL){
                head=temp;
                return *this;
            }
            Node *p=head;
            while(p->next!=NULL)
                p=p->next;
            p->next=temp;
            return *this;
        }
        linkedList &operator -(int val){
            Node *temp=head;
            if(head==NULL)
                return *this;
            if(head->data==val){
                head=head->next;
                delete head;
                return *this;
            }
            while(temp->next && temp->next->data!=val)
                temp=temp->next;
            if(temp->next){
                Node *d=temp->next;
                temp->next=temp->next->next;
                delete d;
            }
            return *this;
        }
        void display(){
            Node *temp=head;
            while(temp){
                cout<<temp->data<<"\t";
                temp=temp->next;
            }
        }
};
int main(){
    linkedList l;
    l=l+1;
    l=l+2;
    l=l+3;
    l.display();
    printf("\n");
    l=l-3;
    l.display();
}
```

Output:

```
1       2       3
1       2
```

Conclusion: The above programs were successfully implemented using operator overloading