**Experiment No: 4 Date:**

**Aim: To study fundamentals of Operator Overloading**

**Theory:**

Operator overloading in C++ allows you to define and redefine the behavior of operators for user-defined data types. This means you can use operators like `+`, `-`, `\*`, `/`, etc., with your own custom classes or data structures.

1. **Syntax**: To overload an operator, you define a function with the keyword `operator` followed by the operator you want to overload. For example, `operator+` overloads the `+` operator.

returnType operator symbol (parameters);

2. **Predefined Operators** : Some operators cannot be overloaded. For example, `.` (member access) and `::` (scope resolution) cannot be overloaded.

3. **Unary vs Binary Operators**:

- Unary operators like `++`, `--`, and `-` take one operand.

- Binary operators like `+`, `-`, `\*`, etc., take two operands.

4. **Return Type**:

- The return type of an overloaded operator depends on the operator being overloaded.

- For example, `operator+` for integers returns an integer, but you can define it to return any valid type.

6. **Overloading for Custom Classes**:

- You can overload operators for any custom class you define.

- This allows you to define meaningful operations for your specific data types.

7. **Operator Overloading and Friend Functions**:

- Sometimes, you might need to access private members of a class in an overloaded operator that involves another object of the same class. In such cases, you can use friend functions.

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

#include<iostream>

using namespace std;

class temp{

int num;

public:

temp(){

num = 0;

}

temp(int \_num){

num = \_num;

}

temp operator ++(){

++ num;

temp cpy(num);

return cpy;

}

temp operator --(){

-- num;

temp cpy(num);

return cpy;

}

temp operator ++(int){

num ++;

temp cpy(num);

return cpy;

}

temp operator --(int){

num --;

temp cpy(num);

return cpy;

}

void display(){

cout<<num<<endl;

}

};

int main(){

temp a(23),b;

++a;

a.display();

a++;

a.display();

--a;

a.display();

a--;

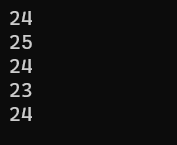
a.display();

b = a++;

b.display();

}

**Output:**



[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

#include<iostream>

using namespace std;

class temp{

int num;

public:

temp(){

num = 0;

}

temp(int \_num){

num = \_num;

}

temp operator \*(temp y){

temp cpy;

cpy.num = num \* y.num;

return cpy;

}

friend temp operator +(int a, temp b){

temp cpy;

cpy.num = a + b.num;

return cpy;

}

temp operator +(int a){

temp cpy;

cpy.num = num + a;

return cpy;

}

void display(){

cout<<num<<endl;

}

};

int main()

{

temp x(20), y(10);

x = 5 + y;

x.display();

x = x \* y;

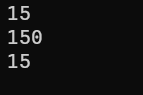
x.display();

x = y + 5;

x.display();

}

**Output:**

****

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

#include<iostream>

using namespace std;

class temp{

int num;

public:

temp():num {0}{}

temp(int n):num {n}{}

friend istream& operator >> (istream &in, temp &n){

in >> n.num;

return in;

}

friend ostream& operator << (ostream &out, temp &n){

out << n.num;

return out;

}

};

int main()

{

temp a,b = 10;

cout<<"Enter a number : ";

cin>>a;

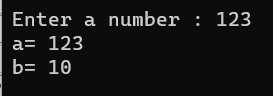
cout<<"a= "<<a<<endl;

cout<<"b= "<<b<<endl;

return 0;

}

**Output:**

****

[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.

#include<iostream>

#include<string.h>

using namespace std;

class STRING{

string string\_content;

public:

STRING():string\_content{}{}

STRING(string str){string\_content = str;}

friend STRING operator +(STRING str\_1, STRING str\_2){

STRING return\_string;

return\_string.string\_content = str\_1.string\_content + str\_2.string\_content;

return return\_string;

}

friend ostream& operator << (ostream &out, STRING &str){

out << str.string\_content;

return out;

}

bool operator ==(STRING str){

return string\_content == str.string\_content;

}

};

int main()

{

STRING a ("Hello "), b ("World") ,c;

cout<<a<<endl;

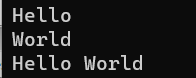
cout<<b<<endl;

c = a + b;

cout<<c;

}

**Output:**

****

[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.

#include<iostream>

#include<stdlib.h>

#include<string.h>

using namespace std;

class Vector{

int \*elements\_ptr;

int len;

int curent\_pos;

public:

//default constructor

Vector():len{0},curent\_pos{0}{

elements\_ptr =NULL;

}

//initialize memory

Vector(int n):len{n},curent\_pos{0}{

elements\_ptr = new int[n];

}

//delete the vector

~Vector(){

delete[] elements\_ptr;

}

//add an element to the vector to at the top position

void push(int element);

//pushes the whole array of elements

void multi\_push(int \*arr,int n);

//overloading = operator to copy a object and uses deep copying

void operator =(Vector vec );

//multiplies all the elements of the vector by a constant

Vector operator \*(int constant);

//adds 2 vectors

Vector operator +(Vector vec);

//display function

string display();

};

void Vector :: push(int element){

if(len == curent\_pos){

if(len == 0)

len = 1;

len\*=2;

int \*tmp\_ptr = new int[len];

for(int i = 0; i<curent\_pos; i++){

tmp\_ptr[i] = elements\_ptr[i];

}

delete[] elements\_ptr;

elements\_ptr = tmp\_ptr;

}

elements\_ptr[curent\_pos] = element;

curent\_pos ++;

}

void Vector :: multi\_push(int arr[], int n = 0){

int i = 0;

while(i<n){

this->push(arr[i]);

i++;

}

}

void Vector :: operator =(Vector vec){

delete[] elements\_ptr;

len = vec.len;

curent\_pos = vec.curent\_pos;

elements\_ptr = new int[len];

for(int i = 0; i<curent\_pos; i++){

elements\_ptr[i] = vec.elements\_ptr[i];

}

}

Vector Vector :: operator \*(int constant){

Vector tmp(len);

for(int i = 0; i < curent\_pos; i++){

tmp.elements\_ptr[i] = elements\_ptr[i] \* constant;

tmp.curent\_pos ++;

}

return tmp;

}

Vector Vector :: operator +(Vector vec){

int i;

Vector tmp;

for (i = 0; i<curent\_pos && i<vec.curent\_pos; i++){

int sum = elements\_ptr[i] + vec.elements\_ptr[i];

tmp.push(sum);

}

while(i <vec.curent\_pos){

tmp.push(vec.elements\_ptr[i]);

i++;

}

while(i <curent\_pos){

tmp.push(elements\_ptr[i]);

i++;

}

return tmp;

}

string Vector :: display(){

string str ="";

for(int i = 0; i<curent\_pos; i++){

str = str + to\_string(elements\_ptr[i]) + ",";

}

str = str + "\0";

return str;

}

int main()

{

Vector a(9),b(5),c;

int arr1[] = {1,2,3,4,5,6,7,8,9};

int arr2[] = {1,3,5,7,9};

a.multi\_push(arr1,sizeof(arr1)/sizeof(arr1[0]));

b.multi\_push(arr2,sizeof(arr2)/sizeof(arr2[0]));

cout<<"Vector 1: "<<a.display()<<endl;

cout<<"Vector 2: "<<b.display()<<endl;

c = a + b;

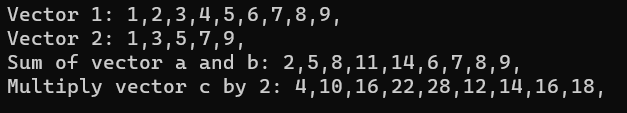
cout<<"Sum of vector a and b: "<<c.display()<<endl;

c = c \* 2;

cout<<"Multiply vector c by 2: "<<c.display()<<endl;

}

**Output:**



**Conclusion:** The concept of operator overloading was undrestood and implemented in the programs above.

**Nitesh Naik**

**(Subject Faculty)**