Unit

6

Operator Overloading

1. Introduction

A programming language provides several operators which can be used to perform operations on values of built-in data types. However, these operators cannot be used on objects of user defined classes. *For example*, + can be used on integers, float etc. But we may want to use + to add two objects of the fraction class. For this, the + operator must be overloaded so that it can be used with fraction objects.

Operator overloading allows us to assign additional meaning to a built-in operator so that they can be used with classes and objects. It is a type of compile-time polymorphism. Most of the standard C++ operators can be overloaded to give them additional functionality.

Table 6.1 lists operators which can be overloaded.

Scanned by CamScanner

Table 6.1: C++ Operators which can be overloaded

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus/assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND/assignment	Binary
( )	Function call	Binary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication/assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment	Unary
+=	Addition/assignment	Binary
–	Subtraction	Binary
–	Unary negation	Unary
– –	Decrement	Unary
– =	Subtraction/assignment	Binary
->	Member selection	Binary

Operator	Name	Type
->*	Pointer-to-member	Binary
/	Division	Binary
/=	Division/assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift/assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift/assignment	Binary
[ ]	Array subscript	Binary
^	Exclusive OR	Binary
^=	Exclusive OR/assignment	Binary
|	Bitwise OR	Binary
|=	Bitwise OR/assignment	Binary
||	Logical OR	Binary
~	Ones complement	Unary
delete	delete	Unary
new	new	Unary

However, there are some operators which **cannot** be overloaded. *Table 6.2* lists these operators.

Table 6.2: C++ Operators which cannot be overloaded

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
sizeof	size information
typeid	type information

▶ Rules for Operator Overloading

1. Only pre-defined, existing operators can be overloaded. The user cannot create new operators. *For example*, we cannot create operator @ to perform operations.
2. Overloading operators does not change the precedence and associativity of the operator.

Scanned by CamScanner

3. Some operators cannot be overloaded. These are member access operator (.) , Pointer-to-member selection (.*) , Scope resolution (::) , Conditional (?:) , sizeof and type conversion operators.

4. The programmer cannot change the syntax or original meaning of an operator. Operators that are binary must remain binary. Unary operators must remain unary. The overloaded operator must have atleast one operand of class type.

5. To overload an operator, define an operator function either as a member function or friend function.

6. The following operators must be overloaded as **members**:

Operator	Name
=	assignment
()	function call
[]	subscript
->	pointer-to-member access

7. The following operators must be overloaded as friends.

Operator	Name
<<	Insertion
>>	Extraction

8. Unary operators overloaded as member functions take no argument. Unary operators overloaded as friend functions take one argument.

9. Binary operators overloaded as member functions take one argument and those overloaded as friend functions take two arguments.

10. When overloaded as member function, the left hand operand must be an object of the class.

11. For overloading the post-increment and post-decrement operators, a dummy argument of type "int" should be used.

2. The Operator Function

To overload an operator, we have to define an operator function. An operator function defines the operation which the overloaded operator will perform. It can be a member of the class or a friend of the class.

Declaration of member operator function

```
return-type operator operator-symbol(parameters);
```

Scanned by CamScanner

Definition of member operator function

```
return-type class-name :: operator operator-symbol(parameters)
{
   //code
}
```

Declaration of friend operator function

```
friend return-type operator operator-symbol(parameters);
```

Definition of friend operator function

```
return-type operator operator-symbol(parameters)
{
   //code
}
```

Depending on the type of operator, the number of arguments passed to the function will vary. The following table shows the number of implicit and explicit arguments for an operator function defined as member or friend. The implicit argument is the object which calls the operator function and the explicit argument is the object which is passed to the function.

Table 6.3: Implicit and explicit arguments for member / friend operator function

	Unary Operator	Binary Operator
Member function	1 implicit, 0 explicit	1 implicit, 1 explicit
Friend function	0 implicit, 1 explicit	0 implicit, 2 explicit

For example, the + (addition) operator requires two objects. If it is overloaded as a member function, we have to pass only one object since the other object will call the function. If it is overloaded as friend, both the objects will have to be passed to the function.

Note When the operator function is defined as a friend function, the members of the object cannot be accessed directly. They have to be accessed using the syntax: object.member

Examples:

1. Overload unary - as member function of fraction class.
   ```
   void operator -();
   ```
2. Overload unary - as friend function of fraction class.
   ```
   friend void operator -(fraction& f);
   ```
3. Overload binary + as member function of fraction class.
   ```
   fraction operator +(fraction& f);
   ```
4. Overload binary + as friend function of fraction class.
   ```
   friend fraction operator +(fraction& f1, fraction& f2);
   ```

Scanned by CamScanner

▶ Invoking an Operator Function

i. Invoking unary operator function

```
operator-symbol object;
```

Example: Member function

```
-f1;        // internal call is  f1.operator -();
```

Example: Friend function

```
-f1;        // internal call is  operator -(f1);
```

ii. Invoking binary operator function

```
object1 operator-symbol object2;
```

Example: Member function

```
f3 = f1-f2;     // internal call is   f3 = f1.operator -(f2);
```

Example: Friend function

```
f3 = f1-f2;     // internal call is   f3 = operator -(f1,f2);
```

3. Overloading Unary Operators

Unary operators are operators which require only one operand. As explained earlier, the operator function can be defined as a member function or a friend function of the class. The following table lists all unary operators in C++ which can be overloaded.

Table 6.4: Unary operators in C++

Operator	Name
!	Logical NOT
&	Address-of
*	Pointer dereference
+	Unary Plus
++	Increment
–	Unary negation
– –	Decrement
~	Ones complement
typename	Type conversion
new	memory allocation
delete	memory deallocation

Scanned by CamScanner

3.1 Overloading as Member Function

1. Overloading Unary - as Member

Syntax:

```
return-type operator -();
```

Example: The following program overloads - for the fraction class as a member function.

Program : Overload unary – operator as member function

```
#include<iostream.h>
class fraction
{
     int numerator, denominator;
     public:
     fraction(int n=0, int d=1)
     {
       numerator=n; denominator=d;
     }
     void display();
     void operator -();    //unary -
};
void fraction::display()
{
  cout<<numerator<<"/"<<denominator<<endl;
}
void fraction::operator -()
{
     numerator = -numerator;
}
int main()
{
  fraction f1(5,4);
  -f1;
  f1.display();
 return 0;
 }
```

Output

-5/4

2. Overloading Increment and Decrement Operators

As you know, the increment and decrement operators can be used in two ways:

1. Pre-increment and post-increment
2. Pre-decrement and post-decrement

For both these cases, the operator function requires only one object. So, to differentiate between pre and post functions, the postfix form is defined with a dummy argument of the type "int". In both cases, the function returns an object of the class.

Scanned by CamScanner

In pre-increment, it should return the incremented object. In post-increment, it should return the old object and also increment the object's value.

Syntax for pre-increment:

```
class-name class-name :: operator ++()
{
  //code
}
```

Syntax for post-increment:

```
class-name class-name :: operator ++(int)
{
  //code
}
```

Note For postfix, the dummy argument must be of type int; specifying any other type generates an error.

Example: The following program overloads ++ for fraction class.

Program : Overload ++ as pre and post increment

```
#include<iostream.h>
class fraction
{
    int numerator, denominator;
    public:
    fraction(int n=0, int d=1)
    {
      numerator=n; denominator=d;
    }
    void display();
    {
        fraction operator ++();    //preincrement
        fraction operator ++(int); //postincrement
    }
};
void fraction::display()
{
  cout<<numerator<<"/"<<denominator<<endl;
}
fraction fraction::operator ++()
{
  numerator = numerator+denominator;
  return *this ; //return implicit object
}
```

Scanned by CamScanner

```
fraction fraction::operator ++(int)
{
   fraction temp(numerator, denominator); //create dummy object
   numerator = numerator+denominator;
   return temp ; //return old value
}
int main()
{
    fraction f1(5,4), f2(5,4), f3;
     f3=++f1;
   cout<<"Preincrement : ";
   f3.display();
   f3=f2++;
   cout<<"Postincrement : ";
     f3.display();
   return 0;
}
```

Output

Preincrement : 9/4
Postincrement : 5/4

3.2 Overloading Unary Operator as Friend Function

To overload a unary operator as a friend, we have to pass the object to the function as parameter. We have seen how unary - is overloaded for fraction class as member. Now let's see how it can be overloaded as a friend function.

Declaration Syntax:

```
friend return-type operator -(class& object);
```

The following code overloads - for the fraction class as a friend function.

```
class fraction
{
    ...
    friend void operator -(fraction& f);
};
void operator -(fraction& f)
{
   f.numerator = -f.numerator;
}
```

Note The difference in the two operator functions: when it is overloaded as a member function, the implicit object can be accessed directly. However, when it is a friend function, the object has to be explicitly passed to the function and its members can be accessed using the dot (.) operator. Since the object is modified by the unary operator, the object should be passed by reference.

Scanned by CamScanner

4. Overloading Binary Operators

In the previous section, we saw how unary operators can be overloaded. Now we will see how to overload binary operators. They can be overloaded as a member function or as friend function.

4.1 Overloading as a Member Function

To overload a binary operator as a member function, we have to pass one argument. The syntax to overload a binary operator as member is:

```
return-type operator operator-symbol(parameters);
```

For example, to overload binary + on fraction objects: fraction operator +(fraction f);

1. **Overloading Arithmetic Operators:** The following *example* overloads the four arithmetic operators **+, -, *** and **/** as member functions on objects of the fraction class and returns the result.

Program: Arithmetic operators overloaded as member functions.

```
#include<iostream.h>
class fraction
{
int numerator,denominator;
public:
   fraction(int n=0, int d=1);  //constructor
void display();
   fraction operator +(fraction&);
   fraction operator -(fraction&);
   fraction operator *(fraction&);
   fraction operator /(fraction&);

};
fraction::fraction(int n, int d)
{
   numerator = n; denominator = d;

}
void fraction::display()
{
   cout<<numerator<< "/"<<denominator<<endl;

}
fraction fraction :: operator +(fraction& f)
{
```

Scanned by CamScanner

```
    fraction temp;
    temp.numerator = numerator*f.denominator + f.numerator*denominator;
    temp.denominator = denominator*f.denominator;
    return temp;
}
fraction fraction :: operator -(fraction& f)
{
   return fraction(numerator*f.denominator-f.numerator*denominator,
   denominator*f.denominator);
}
fraction fraction :: operator *(fraction& f)
{
  return fraction(numerator*f.numerator, denominator*f.denominator);
}
fraction fraction :: operator /(fraction& f)
{
   return fraction(numerator*f.denominator,  denominator*f.numerator)
}
int main()
{
  fraction f1(1,2), f2(3,5), f3;
  cout<<"F1 :";
    f1.display();
  cout<<"F2 :";
    f2.display();
    f3 = f1+f2;
  cout<<"Addition = ";
    f3.display();
    f3 = f1-f2;
  cout<<"Subtraction = ";
    f3.display();
  f3 = f1*f2;
cout<<"Multiplication = ";
  f3.display();
  f3 = f1/f2;
cout<<"Division = ";
  f3.display();
return 0;
}
```

Output

```
F1 : 1/2
F2 : 3/5
Addition = 11/10
Subtraction = -1/10
Multiplication = 3/10
Division = 5/6
```

Scanned by CamScanner

2. **Overloading Relational operators:** The relational operators compare operands and return either true or false. We can overload these to compare objects. *For example:* f1<f2, f1== f2 etc. The operator function will return an int and take one object as explicit argument.

Syntax:

```
int operator operator-symbol(classname&);
```

The following functions overload < and == operators for fraction class. The other operators can be overloaded in the same way.

```
int fraction :: operator <(fraction& f)
{
    if(numerator*f.denominator < denominator* f.numerator)
      return 1;
    return 0;
}
int fraction :: operator ==(fraction& f)
{
    return(numerator*f.denominator == denominator* f.numerator);
}
int main()
{
    fraction f1(1,2), f2(4,8), f3(5,7);
    if(f1==f2)
      cout<<"F1 is equal to F2:"<<endl;
    else
      cout<<"F1 is not equal to F2:"<<endl;
    if(f1<f3)
      cout<<"F1 is less than F3:"<<endl;
    else
      cout<<"F1 is not less than F3:"<<endl;
}
```

3. **Overloading Assignment operator (=)**

Consider the following piece of code:

```
fraction f1(5,4), f2(2,7);
f2=f1;
```

In such a case, the values of data members of f1 will be assigned to f2. This is done by the compiler automatically i.e. the compiler automatically overloads = operator. However, there are some cases when it is important to define an overloaded = operator function for the class. If the class has a pointer member and dynamically allocated memory, it is essential to copy the objects properly.

Scanned by CamScanner

Consider the following class:

```
class Message
{
  char *str;
    int len;
  public:
    Message(); //default constructor
    {
      str=NULL; len=0;
    }
    Message(char *s); //parameterized constructor
    {
      len=strlen(s);
      str=new char[len+1];
      strcpy(str, s);
    }
};
int main()
{
    Message m1("Hello"), m2;
    m2=m1;
}
```

In this *example*, no memory has been allocated for the pointer in m1. The compiler will simply copy the address in m1's pointer to m2's pointer. In such a case, we must overload = so that memory is allocated for m2's pointer and the contents are copied properly.

Syntax:

```
const classname& operator =(const classname&);
```

The right side operand is passed to the function by reference. Its contents will be copied to the left side object which is the implicit object. The implicit object is returned by reference to allow cascading i.e. multiple calls to = operator. *For example:* m3=m2=m1;

The function calls will be given as: m3.operator=(m2.operator=(m1));

Let us now see how = can be overloaded for the string class defined above.

```
class Message
{
  ....
    const Message& operator=(const Message& m)
    {
      len = m.len;
      str = new char[len+1];
      strcpy(str,m.str);
```

Scanned by CamScanner

```
        return *this;
    }
};
int main()
{
  Message m1("Hello"), m2;
   m2=m1;
}
```

Note The assignment operator (=) must be overloaded as a member function.

4. **Overloading Subscript Operator ([ ]):** We have used the subscript operator on an array. It requires two operands: the array and an index. If the class has a group of elements (like an array, file etc) as a data member, then we can overloaded the [] operator for the class. It must be overloaded as a member function.

Syntax:

```
return-type operator [](const int& index);
```

The index will be passed as an explicit argument to the function. The return-type will depend on the type of element stored in the object.

Consider the following class:

```
class Vector
{
    int *ptr;
    int size;
    ...
};
int main()
{
    Vector v1(10); //Vector object of size 10
    cout<<v1[5]; //display the 6th element
}
```

In this case, the [] operator should be overloaded for the Vector class. It can be defined as follows:

```
int Vector::operator[](const int& index)
{
    if(index>=0 && index<size)
      return ptr[index];
    return -999; //invalid number
}
```

Scanned by CamScanner

4.2 Overloading Binary Operator as Friend

To overload a binary operator as a friend function, we have to pass it two arguments. *For example*, to overload binary + on fraction objects:

```
friend fraction operator + (fraction f1, fraction f2);
```

In many cases, whether you overload an operator by using a friend or a member function makes no functional difference. In those cases, it is usually best to overload by using member functions. However, overloading by using a friend increases flexibility of an overloaded operator.

As you know, when you overload a binary operator by using a member function, the object on the left side must be an object of the same class. However, when overloaded as friend, we can pass objects of different types to the operator function.

For example:

```
1.   f1 + f2   //add two fractions
2.   f1 + 5    // add fraction and int
3.   5 + f1    //add int and fraction
4.   Integer obj(10);
     f1 + obj
//add fraction and object of class Integer
```

The functions will be declared as follows:

```
1.   friend fraction operator +(fraction& a, fraction& b);
2.   friend fraction operator +(fraction& f, int n);
3.   friend fraction operator +(int n, fraction& f);
4.   friend fraction operator +(int n, Integer& obj);
```

The following code illustrates how the + operator can be overloaded as a friend for the fraction class.

```
class fraction
{
  int numerator,denominator;
  public:
    friend fraction operator +(fraction& a, fraction& b);
};
fraction operator +(fraction& a, fraction& b)
{
  fraction temp;
  temp.numerator = a.numerator * b.denominator + b.numerator *
  a.denominator;
  temp.denominator = a.denominator*b.denominator;
  return temp;
}
```

Scanned by CamScanner

▶ Overloading Insertion Operator (<<) and Extraction (>>) Operators

The insertion operator and extraction operators have been already overloaded for all built-in types. We can further overload them so that they work in the same manner for user defined class types. *For example*, if s is an object of the class student, then the statements `cin>>s; cout<<s;` should accept and display the student object in an appropriate manner.

These operators should be overloaded as a **friend**. The general format for the function declaration and definition are:

Declaration Syntax:

```
friend  ostream& operator<<(ostream&, const class-name&);
```

```
friend  istream& operator >>(istream& , class-name&);
```

Definition Syntax:

```
ostream& operator<<(ostream& out, const class-name& obj)
{ //display variables of obj using stream object out
  return out;
}
istream& operator >> (istream& in, class-name& obj)
{
  //accept variables of obj using stream object in
  return in;
}
```

The first argument is a reference to the object which calls the function (cout or cin), and it is returned by reference so that function calls can be chained together (cascading). The second argument must be an object of the class. Note that since this is a friend function, we cannot use the "this" pointer.

Example:

```
fraction f1;
cin>>f1;
cout<<"Fraction f1 = "<<f1;
```

The functions are given in the program below.

Program : Overload insertion and extraction operators

```
#include<iostream.h>
class fraction
{
  int numerator,denominator;
  public:
    friend ostream& operator <<(ostream&, const fraction&);
```

Scanned by CamScanner

```
        friend  istream& operator >>(istream& , fraction&);
};
ostream& operator<<(ostream& out, const fraction& f)
{
  out<<f.numerator<<"/"<<f.denominator<<endl;
  return out;
}
istream& operator >>(istream& in, fraction& f) //definition
{
  cout<<"Enter the numerator : ";
  in >> f.numerator;
cout<<"Enter the denominator :";
  in >> f.denominator;
  return in;
}
int main()
{
  fraction f1;
  cin >> f1 ;
  cout << "F1 = "<< f1 ;
  return 0;
}
```

Output

```
Enter the numerator: 3
Enter the denominator: 5
F1 = 3/5
```

4.3 Overloading new and delete Operators

The new and delete operators are memory management operators in C++. Their purpose is to allocate and de-allocate memory dynamically. These operators can be overloaded like other operators.

Syntax for overloading new:

```
void * operator new(size_t);
```

Syntax for overloading delete:

```
void  operator delete(void *);
```

These operators should be overloaded if we wish to add more functionality when allocating or deallocating memory or to allow users to keep track of memory allocation and deallocation in their programs.

Scanned by CamScanner

The operators can be overloaded specifically for a class to control memory allocation and deallocation for objects of that class.

Example:

```
class student
{
   int rollno;
   public:
   student()
   {
     cout<<"Default constructor";
   }
   student(int r)
   {
     rollno=r;
   }
   void * operator new(size_t size)
   {
      cout<<"Overloaded new operator for "<<size<<"bytes "<<endl;
      void *ptr = malloc(size);
      return ptr;
   }
   void operator delete(void *ptr)
   { free(ptr); }
};
int main()
{
   student *ptr = new student(1);
   delete ptr;
   return 0;
}
```

The output of the program will be:

Overloaded new operator for 4 bytes

In the function we have used malloc to allocate memory. We can also use the global new operator for allocating the memory. In this case, the new operator will also invoke the default constructor.

```
void *ptr = ::new student();
```

The output in this case will be:

Overloaded new operator for 4 bytes

Default constructor

Scanned by CamScanner

Solved Examples

1. Create a class FLOAT that contains one float data member. Overload all four arithmeti operators so that they operate on the objects of FLOAT.

```
#include<iostream.h>
class FLOAT
{
   float number;
   public:
   FLOAT operator + (FLOAT n1)
   {
      FLOAT t;
        t.number = number + n1.number;
        return t;
   }
   FLOAT operator - (FLOAT n1)
   {
      FLOAT t;
      t.number = number - n1.number;
      return t;
   }
   FLOAT operator * (FLOAT n)
   {
      FLOAT t;
      t.number = number * n.number;
      return t;
   }
   FLOAT operator / (FLOAT num)
   {
      FLOAT t;
      t.number = number / num.number;
      return t;
   }
   void show()
   {
       cout << number <<endl;
   };
   void in()
   {
      cout << "Enter a number: ";
      cin>>number;
   }
};
void main()
{
    FLOAT ob1, ob2, ob3;
    ob1.in();
    ob2.in();
    ob3 = ob1 + ob2;
    ob3.show();
    ob3 = ob1-ob2;
```

Output

```
Enter a number: 5.2
Enter a number: 3.4
8.6
1.8
17.68
1.529412
```

Scanned by CamScanner

```
ob3.show();
ob3 = ob1 * ob2;
ob3.show();
ob3 = ob1/ob2;
ob3.show();
}
```

2. Write a C++ program to overload the following operators to perform String operations:

i. == Equality
ii. = String Copy
iii. + Concatenation
iv. << To display a String
v. >> To accept a String
vi. - To reverse a string

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#define MAXSIZE 80
class String
{
  private:
    char str[MAXSIZE];
  public:
    String operator + (String&); //concatenation
    String operator = (String&); //assignment
    int operator == (String&); //equality
    String& operator -();      //reverse
    friend ostream& operator << (ostream&, String&);
    friend istream& operator >> (istream&, String&);
};
ostream& operator << (ostream &out, String &s)
{
  out << s.str << endl;
  return out;
}
istream& operator >> (istream &in, String &s)
{
  in.getline(s.str,20);
  return in;
}
String String :: operator + (String &s)
{
  String t;
  strcpy(t.str,str);
  strcat(t.str,s.str);
  return t;
```

Scanned by CamScanner

```
String String :: operator = (String &s)
{
  strcpy(str,s.str);
  return *this;
}
int String :: operator == (String &s)
{ return strcmp(str,s.str)==0;}
String& String :: operator - ()
{
  strrev(str);
  return *this;
}
void main()
{
  String s1,s2,s3;
  int ch;
  while(1)
  {
    clrscr();
    cout << "1. == Equality" << endl
         << "2. = String Copy" << endl
         << "3. + Concatenation" << endl
         << "4. - To reverse a String" << endl
         << "5. Exit" << endl
         << "Enter Your Choice (1-5): ";
    cin >> ch;
    cin.ignore(1,'\n');
    switch(ch)
    {
    case 1:
      cout << "Enter String1: ";
      cin >> s1;
      cout << "Enter String2: ";
      cin >> s2;
      if(s1==s2)
        cout << s1 <<"and"<<s2<< "are equal" << endl;
      else
        cout<<s1<<"and"<<s2<<"are not equal"<< endl;
      break;
    case 2:
        cout <<"Enter String1:";cin>> s1;
          s2 = s1;
        cout<< "Copied String: " << s2 << endl;
      break;
    case 3:
        cout << "Enter String1: ";   cin >> s1;
        cout << "Enter String2: ";   cin >> s2;
```

Scanned by CamScanner

```
        s3 = s1 + s2;
        cout<<"Concatenated String: " << s3 << endl;
      break;
    case 4:
        cout<< "Enter String:"; cin >> s1;
         -s1;
        cout << "Reversed String: " << s1 << endl;
      break;
    case 5: exit(0);
    }
    getch();
  } //end while
}
```

Output

```
Hello
HelloBye
y
```

3. Create a class Message with two data members one character pointer and an integer storing length. Overload operator binary + to represent concatenation of messages, [] to return a character at a specific position and = to copy one Message object to another.

```
#include<iostream.h>
class Message
{
  char *str;
  int len;
public:
  Message() //default constructor
  {
    str=NULL; len=0;
  }
  Message(char *s) //parameterized constructor
  {
    len=strlen(s);
    str=new char[len+1];
    strcpy(str, s);
  }
  const Message & operator =(const Message& m)
  {
    len=m.len;
    str=new char[len+1];
    strcpy(str, m.str);
    return *this;
  }
```

Scanned by CamScanner

```
  Message operator+(const Message& m)
  {
     char *temp;
     temp=new char[len+1];
     strcpy(temp, str);
     len=len+m.len;  //concatenated length
     str=new char[len+1];  //allocate new memory
     strcpy(str, temp);
     strcat(str, m.str);
     return *this;  //return concatenated object
  }
 char operator[](const int& index)
 {
  if(index>=0 && index<size)
  return str[index];
  return '\0';
}
  friend ostream& operator <<(ostream& out, const Message& m)
  {
    out<<m.str<<endl;
    return out;
  }
};
void main()
{
  Message m1("Hello"), m2("Bye"), m3;
  m3=m1;
  cout<<m3;
  m3=m1+m2;
  cout<<m3;
  cout<<m2[1];
}
```

4. Write a C++ class to represent an integer matrix of different dimensions. Overload +, * to perform matrix operations.

```
#include<iostream.h>
class Matrix
{
   int arr[10][10];
   int rows,cols;
   public:
```

Scanned by CamScanner

```
Matrix(int r=5, int c=5)
{
  rows=r; cols=c;
}
void accept()
{
  int i,j;
  cout<<"Enter "<<rows<<"*"<<cols<<" elements :"<<endl;
  for(i=0; i<rows; i++)
    for(j=0; j<cols; j++)
      cin>>arr[i][j];
}
void display()
{
  int i,j;
  cout<<"The matrix is :"<<endl;
  for(i=0; i<rows; i++)
  {
    for(j=0; j<cols; j++)
      cout<<" "<<arr[i][j];
    cout<<endl;
  }
}
Matrix operator+(const Matrix& m)
{
  int i,j;
  if((rows==m.rows)&&(cols==m.cols))
  {
    Matrix temp(rows, cols);
    for(i=0; i<rows; i++)
    for(j=0; j<cols; j++)
      temp.arr[i][j]=arr[i][j]+m.arr[i][j];
    return temp;
  }
  else
  {
    cout<<"Matrices could not be added";
    return Matrix(0,0);
  }
}
Matrix operator*(const Matrix& m)
{
  int sum, i, j, k;
  if(cols==m.rows)
  {
    Matrix temp(rows, m.cols);
    for(i=0; i<rows; i++)
```

Scanned by CamScanner

```
            for(j=0; j<m.cols; j++)
            {
               sum=0;
               for(k=0; k<cols; k++)
               sum=sum+arr[i][k]*m.arr[k][j];
               temp.arr[i][j]=sum;
            }
            return temp;
         }
         else
         {
            cout<<"Matrices could not be multiplied";
            return Matrix(0,0);
         }
      }
};
void main()
{
   Matrix a(3,3), b(3,3), c(3,3);
   a.accept();
   b.accept();
   c=a+b;
   c.display();
   c=a*b;
   c.display();
}
```

EXERCISES

A. Multiple Choice Questions.

1. From the following list which operator cannot be overloaded.
   a. Conditional operator b. Comma operator
   c. Division operator d. Assignment operator

2. Which of the following cannot be overloaded as a member function?
   a. ++ b. =
   c. << d. [ ]

3. How many implicit arguments will a binary operator function overloaded as friend take?
   a. 0 b. 1
   c. 2 d. None of the above

Scanned by CamScanner

4. Which statement is false about operator overloading:
   a. New operators cannot be created.
   b. The precedence of operators cannot be changed.
   c. We can change the number of operands an operator takes by overloading.
   d. We can define operator functions as friends.

5. Which is the correct syntax to overload the ++ operator for postincrement :
   a. ++ operator()
   b. ++ operator (int)
   c. operator ++()
   d. operator ++(int)

6. Which is the correct syntax to overload the << operator for class Data
   a. `ostream operator<<(ostream &out, const Data &obj)`
   b. `ostream operator<<( const Data &obj, ostream &out)`
   c. `ostream &operator<<(ostream &out, const Data &obj)`
   d. `ostream &operator<<( const Data &obj, ostream &out)`

B. State True or False.

1. Operator overloading allows us to assign additional meanings to C++ operators.
2. When we overloading operators the precedence and associativity of the operator changes.
3. The overloaded operator must have at least one user defined type operand.
4. All operators in C++ can be overloaded.
5. The insertion and extraction operators have to be overloaded as members.

C. Review Questions.

1. What is meant by operator overloading?
2. List the C++ operators that cannot be overloaded.
3. Explain the limitations of operator overloading.
4. Explain the rules of operator overloading.
5. Explain how the preincrement and postincrement operators are overloaded.
6. Explain the syntax of overloading the insertion and extraction operators.
7. Explain how the typecast operator can be overloaded for a class.

Scanned by CamScanner

D. Programming Exercises

1. Use the following definition of the C++ class Currency:

```
class Currency
{
private:
  long int totalpaise;
public:
  Currency(long int rup = 0, int paise = 0);
  long int rupees() const;
  int paise() const;
  Currency& operator=(const Currency&);
  Currency& operator+=(const Currency&);
  Currency& operator-=(const Currency&);
  friend ostream& operator<<(ostream&, const Currency&);
  friend istream& operator>>(istream&, Currency&);
  friend const Currency operator+(const Currency&, const Currency&);
  friend const Currency operator-(const Currency&, const Currency&);
};
```

   i. Implement the Currency class with the operators and the additional global operators as shown

   ii. Create some Currency values and add/subtract/assign them to each other to test out your operators.

   iii. Test your implementation of operator<< and operator>> by printing some currency values to cout and reading a Currency from cin.

2. Define a class for a matrixOverload the following operators : + for addition, - for subtraction, - for negation, == for equality, << for displaying, >> for accepting, = for assignment. Write a menu driven program to implement the above.

3. Define a class "MyString" with a string data member. Overload the following operators to perform operations on the objects: = (copy one object to another), << >> (insertion and extraction), == (compare), binary + (Concatenate), unary - (change case), binary * (intersection)

Answers

A.

1.	a,c	2.	a
3.	c	4.	c
5.	c	6.	d

B.

1.	True	2.	False		
3.	True	4.	False	5.	False

Scanned by CamScanner