**Introduction :**

* **What are Unary Operators?**

The Operators which operate on Single Operand known as Unary Operators, some of the unary operators are:

|  |
| --- |
| **++ Increment Operator**  **-- Decrement Operator**  **& Address Of Operator**  **- Unary Minus Operators**  **~ (One’s Compliment) Negation Operator**  **! Logical NOT** |

* **What are Binary Operators?**

The Operators which operate on Two Operands known as Binary Operators, some of the binary operators are:

|  |
| --- |
| **+ Binary Plus Operator**  **- Binary Minus Operator**  **== Equal to Operator**  **< Less than Operator** |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**DEFINING OPERATOR OVERLOADING**

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. An Operator function is a special function which describes the task.

The general form of an operator function is:

**return-type class-name :: operator op(argument-list)**

**{**

**function body**

**}**

where return type is the type of value returned by the specified operation and op is the operator being overloaded. ***The op is preceded by the keyword operator.*** ***Operator op is the function name.***

Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.

|  |
| --- |
| FRIEND function - one argument for unary operators  two for binary operators  MEMBER function has no arguments for unary operators and only one for binary operators. |

**OVERLOADING UNARY OPERATORS**

The minus operator changes the sign of an operand when applied to a basic data item. The unary minus when applied to an object should change the sign of each of its data items.

Example:

class single

{

int x, y, z;

public:

void getdata(int, int, int);

void display(void);

void operator-(); //overloaded unary minus

};

void single::getdata(int a, int b, int c)

{ x=a; y=b; z=c;

}

void example:: display(void)

{ cout<< “x=”<< x <<”\t”<<”y=”<< y <<”\t” << “z=”<< z <<endl;

}

void single :: operator- ()

{

x= -x;

y= -y;

z= -z;

}

void main()

{

single S;

S.getdata(10, -20, 30);

cout<<”Entered Numbers are: “<<endl;

S.display();

-S;

cout<<”After calling unary minus operator:”<<endl;

S.display();

}

OUTPUT:

Entered Numbers are:

x=10 y= -20 z=30

After calling unary minus operator:

x= -10 y= 20 z= -30

The function operator –() takes no argument. It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the member of the object which activated it.

It is possible to overload a unary minus operator using a friend function as follows:

1. Replace the member function declaration by the friend function declaration

friend void operator – (single &s); //declaration

1. Redefine the operator function as follows:

void operator – (single &s)

{

s.x = - s.x;

s.y = - s.y;

s.z= - s.z;

}

Note that the argument is passed reference, because only a copy of the object that activated the call is passed to operator (-). Therefore, the changes made inside the operator function will not reflect in the called object.(\*refer class notes)

**OVERLOADED BINARY OPERATORS USING MEMBER FUNCTIONS**

Example to find the sum of two complex numbers using overloaded binary operators using member functions.

class complex

{

float real,imag;

public:

void getdata(float r, float i)

{ real=r;

imag=i;

}

void display(void)

{ cout<<real <<"+j"<<imag<<endl;

}

complex operator + (complex);

};

complex complex:: operator + (complex c)

{ complex c3;

c3.real=real+c.real;

c3.imag=imag+c.imag;

return c3;

}

main()

{ complex A,B,C;

A.getdata(2.3,4.5);

B.getdata(3.6,1.1);

C=A+B;

cout<<"A=";A.display();

cout<<"B=";B.display();

cout<<"SUM=";C.display();

}

We should note the following features of this function:

1. It receives only one complex type argument explicitly.
2. It returns a complex type value.
3. It is a member function of complex.

Here the statement

C=A+B;

invokes the operator function. Here the object A takes the responsibility of invoking the function and B plays the role of an argument that is passed to the function. The above invocation statement is equivalent to

C=A.operator+(B);

Therefore, in the operator+() function, the data members of A are accessed directly and the data members of of B( that is passed as an argument) are accessed using the dot operator. Thus both the objects are available for the function. As a rule, in overloading of binary operators, the left hand operand is used to invoke the operator function and the right hand operand is passed as an argument.

**OVERLOADED BINARY OPERATORS USING FRIEND FUNCTIONS**

Overloading of binary operators using friend functions requires two arguments to be explicitly passed to it. The above program can be modified using a friend function as follows:

1. Replace the member function declaration by the friend function declaration.

friend complex operator+(complex, complex);

1. Redefine the operator function as follows:

complex operator+(complex c1, complex c2)

{

complex c3;

c3.real=c1.real+c2.real;

c3.imag=c1.imag+c2.imag;

return c3;

}

Example to find the sum of two complex numbers using overloaded binary operators using friend functions.

class complex

{

float real,imag;

public:

void getdata(float r, float i)

{

real=r;

imag=i;

}

void display(void)

{ cout<<real <<"+j"<<imag<<endl;

}

friend complex operator+(complex, complex);

};

complex operator+(complex c1, complex c2)

{ complex c3;

c3.real=c1.real+c2.real;

c3.imag=c1.imag+c2..imag;

return c3;

}

main()

{ complex A,B,C;

A.getdata(2.3,4.5);

B.getdata(3.6,1.1);

C=A+B;

cout<<"A=";A.display();

cout<<"B=";B.display();

cout<<"SUM=";C.display();

}

[ Example to find the sum of two distances using overloaded binary operators using friend functions \* refer class notes ]

**MANIPULATION OF STRINGS USING OPERATORS**

In C the string operations are not possible using the statements like,

string3=string1+string2;

if (string1>=string2) string= string1;

But it is possible in C++. In C++, to manipulate strings, a new class called string is created. Strings can be defined as class objects.

CONCATENATING STRINGS: Example program to concatenate two strings using operators overloading

class string

{

char str[20];

public:

void input(void);

string operator+(string);

void output(void);

};

void string::input(void)

{

cin>>str;

}

string string ::operator+(string s2)

{

string s3;

strcpy(s3.str,str);

strcat(s3.str,s2.str);

return(s3);

}

void string::output()

{

cout<<str<<endl;

}

main()

{

string s1,s2,s3;

cout<<"Enter a string1:";

s1.input();

cout<<"Enter a string2:";

s2.input();

s3=s1+s2;

cout<<endl<<"Concatenated string is :";

s3.output();

}

OUTPUT:

Enter a string1: SBC

Enter a string2:karkala

Concatenated string is :SBCkarkala

COMPARING STRINGS: Example program to concatenate two strings using operators overloading.

class string

{ char str[20];

public:

void input(void);

int operator==(string s);

int operator<(string s);

void output(void);

};

void string::input(void)

{ cin>>str;

}

int string::operator==(string s)

{ if (strcmp(str,s.str)==0)

return 1;

else

return 0;

}

int string::operator<(string s)

{ if(strcmp(str,s.str)<0)

return 1;

else

return 0;

}

void string::output(void)

{ cout<<str<<endl;

}

main()

{

string s1,s2,s3;

cout<<"Enter string1:";

s1.input();

cout<<"Enter string2:";

s2.input();

if(s1==s2)

cout<<"Strings are equal"<<endl;

else

cout<<"Strings are not Equal"<<endl;

if(s1<s2)

cout<<"s1 is smaller than s2"<<endl;

else

cout<<"s1 is greater than s2"<<endl;

}

OUTPUT:

Enter string1:KARKALA

Enter string2:KERALA

Strings are not Equal

s1 is smaller than s2

**Rules for overloading operators:**

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus( +) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. (i.e. Sizeof, . , .\* , :: , ?: )
6. We cannot use friend functions to overload certain operators. (i.e. = , ( ) , [ ] , →) However, member functions can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, -, \*, and / must explicitly return a value. They must not attempt to change their own arguments.

**TYPE CONVERSIONS**

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

int m;

float x =3.14159;

m = x;

convert x to an integer before its value is assigned to m. Thus, the fractional part is truncated. The type conversions are automatic as long as the *data types involved are built-in types.*

What happens when they are user-defined data types?

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type

**BASIC TO CLASS TYPE**

The conversion from basic type to class type is easy to accomplish. To convert a basic type to a class type, we should create a constructor, where constructor perform a *defacto* type conversion from the argument's type to the constructor's class type.

Example: CONVERSION FROM BASIC TYPE TO CLASS TYPE

class time

{

int h,m;

public:

void output(void)

{ cout<<h<<" Hours "<<m<<" Minutes ";

}

time(){}

time(int t)

{

h=t/60;

m=t%60;

}

};

void main()

{

time T;

int duration=85;

T=duration;

T.output();

}

This program converts an int type to a class type.

The statement

T=duration;

invokes the constructor implicitly.

**CLASS TO BASIC TYPE**

To convert a class type to basic type , define an overloaded casting operator that could be used to convert a class type to a basic type. The general form of an overloaded casting operator function usually referred to as a conversion function, is:

operator typename()

{ ………………….

Function statements

………………….

}

The casting operator function should satisfy the following conditions:

* It must be a class member.
* It must not specify a return type.
* It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, conversion inside the function belongs to the object that invoked the function. This means that the function not needs an argument.

Example: CONVERSION FROM CLASS TYPE TO BASIC TYPE

class time

{ int h,m;

public:

time()

{ h=2;

m=10;

}

operator int()

{ h=h\*60;

m=m+h;

return m;

}

};

void main()

{

time T;

int duration;

duration=T;

cout<<"The total minutes="<<duration;

}

**ONE CLASS TO ANOTHER CLASS TYPE**

Example: objX =objY // objects of different classes

objX is an object of class X and objY is an object of class Y. The class Y type data is converted to the class X type data and the converted value is X. Since the conversion takes place from class Y to class X, Y is known as source class and X is known as destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Then, how to decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

operator typename()

In the case of conversion between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a constructor function which converts argument’s type to the class type. This implies that the argument belongs to the source class and is placed in the destination class.

It shows that the conversion from a class to any other type(or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type /class to a class type, a constructor should be used in the destination class.

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

[ Write a program to use two classes and shows how to convert data of one type to another \* refer class note