Operator Overloading

Introduction

- Operator overloading
  - ➢ Enabling C++’s operators to work with class objects
  - ➢ Using traditional operators with user-defined objects
  - ➢ Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
    - Operator **>>** is both the stream-extraction operator and the bitwise right-shift operator
  - ➢ Compiler generates the appropriate code based on the manner in which the operator is used.

Introduction (Cont…)

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **operator** followed by the symbol for the operator being overloaded
  - For e.g., **operator+** used to overload the addition operator (**+**)
- Using operators
  - To use an operator on a class object, it must be overloaded unless the assignment operator **(=)** or the address operator **(&)**
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - number of operands cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators **&**, *****, **+** and **–** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Restrictions on Operator Overloading

- All the C++ operators can be overloaded except the following:
  - ➢ Scope Resolution Operator (::)
  - ➢ Conditional Operator (? :)
  - ➢ Size Operator (sizeof)
  - ➢ Class Member Access Operators (. and .*)

Operator Functions as Class Members vs. as friend Functions

- Operator functions can be either member or non-member functions of a class.
- Operator functions as member functions
  - Leftmost operand must be an object of the class
- Operator functions as non-member functions
  - Must be `friend functions` of the class
  - Enables the operator to be commutative

Note: *If left operand is of a different type than class, operator function must be a non-member function*

Creating a Member Operator Function

```
returntype classname::operator opname(arg-list)
{
    // operations
}
```

Keyword

Unary Operators

- The unary operators operate on a single operand and following are the examples of Unary operators –

- ➢ The increment (++) and decrement (--) operators.
- ➢ The unary minus (-) operator.
- ➢ The logical not (!) operator.

Overloading of Unary Operator - (using member function)

```
#include <iostream>
using namespace std;
class Distance
 {
 private:
  int feet;
  int inches;
public:
//Default constructor
Distance();
//Parameterized constructor
Distance(int , int )
// method to display distance
void distance::displayDistance() ;
// overloaded unary minus (-) operator
Distance operator- ()
};
```

```
Distance::Distance()
{
 feet = 0;
  inches = 0;
 }

Distance::Distance(int f, int i)
 {
 feet = f;
 inches = i;
  }

void distance::displayDistance()
{
cout << "F: " << feet << " I:" << inches <<endl;
 }
```

Continued…

```
Distance Distance::operator- ()
{
  feet = -feet;
  inches = -inches;
  return Distance(feet, inches);
 }

int main()
 {
     Distance D1(11, 10), D2(-5, 11);
     -D1;          // activates operator-() function
      D1.displayDistance(); // display D1
      -D2;           // activates operator-() function
      D2.displayDistance();    // display D2
      return 0;
}
```

Output:

```
F: -11 I:-10 F: 5 I:-11
```

Binary Operators

- The binary operators take two arguments.

- Binary operators are used very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Overloading Binary Operators

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
//constructors
loc() {
}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show()
{
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};

// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
```

Continued…

```
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
   /*equivalent to
    ob1=ob1.operator+(ob2);*/
ob1.show(); // displays 15 50
return 0;
}
```

Output:

```
10  20
5   30
15  50
```

Overloading +, -(binary) and ++(unary)

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // constructors
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
```

Continued…

```
// Overload - for loc
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload asignment for loc
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload prefix ++ for loc
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
```

```
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
ob2.show(); // displays 90 90
return 0;
}
```

Output:

```
10      20
5       30
11      21
12      22
12      22
90      90
```

Overloading using Friend Functions

- There are certain situations where we would like to use a friend function rather than a member function.
- For example, if we want to use two different types of a binary operator, say, one an object and another a built in type as shown below:

  A=B+2, where A and B are objects of same class. This will work for a member function, but the statement **A=2+B**; will not work.
- This is because the left-handed operand which is responsible for invoking the membership function should be an object of the same class.
- Friend function allows both approaches as it is invoked without the use of objects.

Continued…

- Unary operators, overloaded by means of a member function, take no explicit arguments, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).

- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

- When using binary operators overloading through member function, the left hand operand must be an object of the relevant class.

Example

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // constructors
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2);
 // now a friend
};
/*Now, + is overloaded using friend function*/
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
```

Continued…

```
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
return 0;
}
```

Output:

```
15  50
```

Restrictions on application of friend function

- Operators that can not be overloaded using a friend function are:

=	Assignment operator
( )	Function call operator
[ ]	Subscript operator
->	Class Member Access Operator

Overloading new and delete

```
// Allocate an object
void *operator new(size_t
    size)
{
/* Perform allocation.
 Throw bad_alloc on failure.
Constructor called
    automatically. */
return pointer_to_memory;
}
```

```
// Delete an object
void operator delete(void
    *p)
{
/* Free memory pointed to
    by p.
Destructor called
    automatically. */
}
```

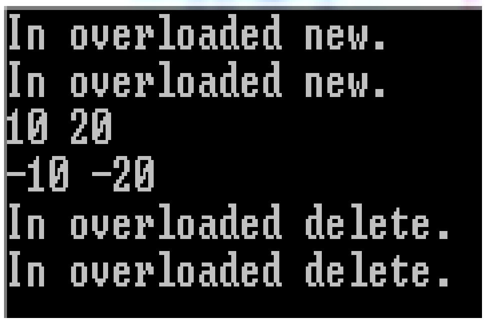
Example

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt)
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
void *operator new(size_t size);
void operator delete(void *p);
};
```

```
// new overloaded relative to loc
void *loc::operator new(size_t size)
{
void *p;
cout << "In overloaded new.\n";
p = malloc(size);
return p;
}
// delete overloaded relative to loc
void loc::operator delete(void *p)
{
cout << "In overloaded delete.\n";
free(p);
}
```

Continued…

```
int main()
{
loc *p1, *p2;

p1 = new loc (10, 20);
p2 = new loc (-10, -20);
}
p1->show();
p2->show();
delete p1;
delete p2;
return 0;
}
```



```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

Overloading [ ] operator

- In C++, the [ ] is considered a binary operator when you are overloading it.

- Therefore, the general form of a member operator[ ]( ) function is as shown here:

```
returntype classname::operator[](int i)
        {
                // . . .
        }
```

- Technically, the parameter does not have to be of type int, but an operator[ ]( ) function is typically used to provide array subscripting, and as such, an integer value is generally used.

- Given an object called obj, the expression obj[3] translates into this call to the operator[ ]( ) function: obj.operator[](3)

Example 1

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int operator[](int i)
 { return a[i];
}
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

Output:

2

Example 2

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i) {
return a[i];
 }
};
```

```
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] on left of =
cout << ob[1]; // now displays 25
return 0;
}
```

Output:

```
2   25
```

Overloading ()

- When we overload the ( ) function call operator, we are not creating a new way to call a function.
- Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Example:

double operator()(int a, float f, char *s);

and an object O of its class, then the statement

O(10, 23.34, "hi");

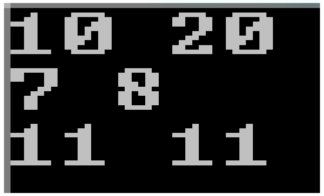
translates into this call to the operator( ) function.

O.operator()(10, 23.34, "hi");

Example

```cpp
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt)
{
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator()(int i, int j);
};
// Overload ( ) for loc
loc loc::operator()(int i, int j)
{
longitude = i;
latitude = j;
return *this;
}
```

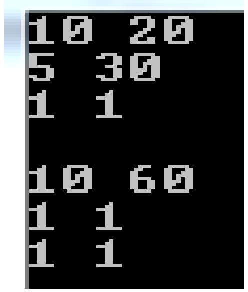
```cpp
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}


int main()
{
loc ob1(10, 20), ob2(1, 1);
ob1.show();
ob1(7, 8); // can be executed by itself
ob1.show();
ob1 = ob2 + ob1(10, 10); // can be used in expressions
ob1.show();
return 0;
}
```



```
10  20
7  8
11  11
```

Overloading the Comma Operator

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator,(loc op2);
};
// overload comma for loc
loc loc::operator,(loc op2)
{
loc temp;
temp.longitude = op2.longitude;
temp.latitude = op2.latitude;
cout << op2.longitude << " " << op2.latitude << "\n";
return temp;
}
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
ob1.show();
ob2.show();
ob3.show();
cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1, the value of ob3
return 0;
}
```



```
10 20
5 30
1 1

10 60
1 1
1 1
```

Overloading ->

- The –> pointer operator, also called the *class member access operator, is considered* a unary operator when overloading.

- Its general usage is shown here:

  object->element;

- Here, *object is the object that activates the call. The operator–>( ) function must return* a pointer to an object of the class that operator–>( ) operates upon.

- The *element must be* some member accessible within the object.

- The example illustrates overloading the –> by showing the equivalence between ob.i and ob–>i when operator–>( ) returns this pointer:

Example

```
#include <iostream>
using namespace std;
class myclass {
public:
int i;
myclass *operator->()
{
return this;
}
};
```

```
int main()
{
myclass ob;
ob->i = 10;
// same as ob.i
cout << ob.i << " " << ob->i;
return 0;
}
```

Output:

```
10    10
```

Type Conversions

- When constants and variables of different types are mixed in an expression, C++ applies automatic type conversion to the operand as per certain rules. Similarly, an assignment operator also causes the automatic type conversion.
- 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?

❑ What if one of the operands is an object and the other is built-in type variable?

❑ What if they belong to two different classes?

Continued…

- Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types.

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

- ➢ Conversion from basic type to class type
- ➢ Conversion from class type to basic type
- ➢ Conversion from class type to class type

Basic to Class Type

- In this type of conversion, the source type is basic type and the destination type is class type, i.e. basic data type is converted into the class type.

- The conversion from basic type to the class type can be performed in two ways:

- ➢ Using constructor
- ➢ Using Operator Overloading

Example (Using Constructor)

```
// Program to convert basic type to class type using constructor
#include <iostream>
using namespace std;
class Time {
 int  hrs, min;
     public:
   Time (int);
    void display();
};
Time :: Time (int t)
{
cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
hrs=t/60;
min=t%60;
}
void Time::display()
{ cout<<hrs<< ": Hours(s)" <<endl; cout<<min<< " Minutes" <<endl;
}
```

```
int main()
{
int  duration;
cout<<"Enter time duration in minutes";
cin>>duration;
Time t1=duration;
t1.display();
return 0;
}
```

During type conversion using the constructor we can pass only one argument and we can do type conversion at the type of initialization only.

Using Operator Overloading

- Type conversion from basic to class type can also be done by operator overloading.

- Assignment operator can be overloaded for this purpose.

- Previous example of *Time* class can be rewritten for type conversion using operator overloading concept to overload the assignment operator (=).

Example

```
// Program to convert from basic type to class type using operator overloading
 #include<iostream>
using namespace std;
 class Time {
   int hrs, min;
 public:
void display();
void operator=(int);              // overloading function
 };
void Time::display()
 {
cout<<hrs<< ": Hour(s) "<<endl ; cout<<min<<": Minutes"<<endl ;
}
void Time::operator=(int t)
{
cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
hrs=t/60;
 min=t%60;
 }
```

Continued…

```
int main()
{
Time t1;
 int duration;
cout<<"Enter time duration in minutes";
cin>>duration;
cout<<"object t1 overloaded assignment..."<<endl;
t1=duration; //or, t1.operator=(duration);
t1.display();
return 0;
}
```

By using overloaded assignment operator we can perform the type conversion at any place in program.

Class to Basic Type

- In this type of conversion the source type is class type and the destination type is basic type, i.e. class data type is converted into the basic type.
- The constructor functions do not support this operation.
- It requires special casting operator function.
- The syntax for an overloaded casting operator function, usually referred to as a conversion function, is:

```
operator typename( )
{
   //Function statements
 }
```

Continued…

- The conversion function should satisfy the following condition:

- ➢ It must be a class member.
- ➢ It must not specify the return value.
- ➢ It must not have any argument.

Example

```
#include<iostream>
using namespace std;
 class Time
{
  int hrs, min;
public:
  Time (int ,int); // constructor
   operator int();             // casting operator function
   ~Time()        // destructor
    {
      cout<<"Destructor called..."<<endl;
    }
};
Time::Time (int a,int b)
{
cout<<"Constructor called with two parameters..."<<endl;
hrs=a;
min=b;
 }
```

Continued…

```
Time :: operator int()
 {
cout<<"Class Type to Basic Type Conversion..."<<endl;
return(hrs*60+min);
 }
int main()
{
int h, m, duration;
cout<<"Enter Hours ";
cin>>h; cout<<"Enter Minutes ";
cin>>m;
Time t(h,m);      // construct object
duration = t;     // casting conversion OR duration = (int)t
cout<<"Total Minutes are "<<duration;
cout<<"2nd method operator overloading "<<endl;
duration = t.operator int();
 cout<<"Total Minutes are "<<duration;
return 0;
}
```

Class to Class Type

- In this type of conversion both the type that is source type and the destination type are of class type.

- Conversion from one class to another class can be performed either by using the ***constructor*** or ***type conversion function***.

Example

```
//Program to convert class Time to another class Minute
#include<iostream>
using namespace std;
class Time
{
  int hrs, min;
public:
   Time(int h,int m)
    {
    hrs=h;
    min=m;
    }
    Time()
    {
    cout<<"\n Time's Object Created";
    }
int getMinutes()
{
int tot_min = ( hrs * 60 ) + min ;
return tot_min;
 }

 void display()
 {
 cout<<"Hours: "<<hrs<<endl ;
 cout<<" Minutes : "<<min <<endl ;
 }
};
```

Continued…

```
class Minute
 {
  int min;
public:
  Minute()
  {
  min = 0;
   }
   void operator=(Time T)
    {
      min=T.getMinutes();
    }
     void display()
      {
      cout<<"\n Total Minutes : " <<min<<endl;
     }
 };
```

```
int main()
{
Time t1(2,30);
t1.display();
Minute m1;
m1.display();
m1 = t1;  // conversion from Time to Minute
 t1.display();
 m1.display();
return 0;
}
```

Continued…

```
int main()
{
Time t1(2,30);
t1.display();
Minute m1;
m1.display();
m1 = t1;        // conversion from Time to Minute
 t1.display();
 m1.display();
return 0;
}
```

Type Conversions Summary

Conversion required	Conversion takes place in	
	Source Class	Destination Class
Basic to Class	Not Applicable	Constructor
Class to Basic	Casting Operator	Not Applicable
Class to Class	Casting Operator	Constructor

Thanks