Operator Overloading

Prepared

by

Naveen Choudhary

Operator Overloading

```
ret_type classname::operator # (arg_list)
{
        //operations
}
```

→when you are overloading a unary operator, arg_list will be empty. when you are overloading binary operators, arg_list will contain one parameter.

```
#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);
};
// 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);
  ob1.show(); // displays 10 20
  ob2.show(); // displays 5 30
  ob1 = ob1 + ob2;
  ob1.show(); // displays 15 50
  return 0;
}
```

Operator overloading of +, - , =, ++ (prefix)

Note:- In C++, if the = is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member-by-member bitwise copy. By overloading the =, you can define explicitly what the assignment does relative to a class. In this example, the overload = does exactly the same thing as the default, but in other situations, it could perform other operations.

```
#include <iostream>
using namespace std;
class loc {
  int longitude, latitude;
public:
  loc() {} // needed to construct
           //temporaries
  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++();
};
```

Prepared by Dr. Naveen Choudhary

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

Creating prefix & postfix forms of ++ & -- operators

if ++ precedes its operand (++ob) → operator ++() // function is valid
if ++ follows its operand (ob++) → operator++(int x)// function is called,
//where x is a dummy argument and has the value 0.

prefix

```
type operator ++ ( )
{
        -----
}
```

Postfix

```
type operator ++ (int x)
{
        -----
}
```

Overloading short hand operators like +=, -=

```
loc loc::operator += (loc op)
{
        longitude = op.longitude + longitude;
        latitude = op.latitude + latitude;
        return *this;
}
```

Operator overloading restrictions & friend function

Operator overloading restrictions

→Using operator overloading, you can not alter the precedence of an operator

→You can't change the no. of operands that an operator takes

→Operation functions can't have default arguments (except for overloaded function call operator)

→The . , :: , .*, ? operator can't be overloaded

Operator overloading using friend function

Since a friend function is not a member of the class, it does not have a this pointer, therefore an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters and a friend function that overloads a unary operator has one parameter. when overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

Operator overloading using friend function

```
#include <iostream>
using namespace std;
class loc {
  int longitude, latitude;
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }
  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
  friend loc operator+(loc op1, loc op2); // friend
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};
// 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;
}
```

```
// 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 assignment for loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;
  return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
  longitude++;
  latitude++;
  return *this;
}
int main() {
  loc ob1(10, 20), ob2( 5, 30);
  ob1 = ob1 + ob2;
  ob1.show();
  return 0;
}
```

Restriction on friend operator function

=, (), [] and → can not be overloaded using friend functions.
→ if ++ or -- are overloaded using friend function then we will need to use reference parameters {pass by reference is necessary because the change (due to ++) should be reflected in the actual object}

```
#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);
  friend loc operator++(loc &op);
  friend loc operator--(loc &op);
// friend, postfix version of ++
//friend loc operator++(loc &op, int x);
};
```

```
// Overload assignment for loc.
loc loc::operator=(loc op2) {
  longitude = op2.longitude;
  latitude = op2.latitude;
  return *this; // i.e., return object
                //that generated call
}
// Now a friend; use a reference
//parameter.
loc operator++(loc &op) {
  op.longitude++;
  op.latitude++;
  return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op) {
  op.longitude--;
  op.latitude--;
  return op;
}
```

```
int main() {
  loc ob1(10, 20), ob2;
  ob1.show();
  ++ob1;
  ob1.show(); // displays 11
              // 21
  ob2 =  ++ob1;
  ob2.show(); // displays 12
              // 22
  --ob2;
  ob2.show(); // displays 11
              // 21
  return 0;
}
```

```
//for post increment in friend
//function a extra dummy
//argument will be reqired
```

Where friend is a must ?

object + integer } ob + 100;
integer + object } 100 + ob;
// overloaded operator member function can not work, so in such cases friend operator
// overloaded function is necessary.

```
#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";
  }
  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};
// + is overloaded for loc + int.
loc operator+(loc op1, int op2) {
  loc temp;
  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;
  return temp;
}
```

```
// + is overloaded for int + loc.
loc operator+(int op1, loc op2) {
  loc temp;
  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;
  return temp;
}
int main() {
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);
  ob1.show();
  ob2.show();
  ob3.show();
  ob1 = ob2 + 10; // both of these
  ob3 = 10 + ob2; // are valid
  ob1.show();
  ob3.show();
  return 0;
}
```

Overloading new & delete

allocate an object

void * operator new (size_t, size)
size →no. of bytes to be allocate {\\ size_t (in #include<new>)is a defined type capable of containing the largest single piece of memory that can be allocated }

1. Perform allocation. Throw bad_alloc on failure

2. Constructor called automatically { no explict code required }

3. return pointer_to_memory {pointer to allocated memory}

Delete an object

```
void operator delete (void *p)  //pointer to the memory to be freed
{
    /*1. Free memory pointed to by p
    2. Destruction called automatically */
}
```

Note:- The *new* and *delete* operators may be overloaded globally so that all users of these operators call your custom versions. they may also be overloaded relative to one or more class. The example below overloads *new* & *delete* relative to a class

Overloading new & delete

```
#include <iostream>
#include <cstdlib>
#include <new>
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);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}
// delete overloaded relative to loc.
void loc::operator delete(void *p) {
  cout << "In overloaded delete.\n";
  free(p);
}
int main()
{
  loc *p1, *p2;
  try {
    p1 = new loc (10, 20);// float *f = new float;
                          // uses default new
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;
  }
  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }
  p1->show();
  p2->show();
  delete p1;
  delete p2;
  return 0;
}
```

Overloading new & delete

Note:- When *new* or *delete* are encountered, the compiler first checks to see whether they are defined relative to the class they are operating on. if so, those specific versions are used. if not, C++ uses the globally defined new & delete. if these have been overloaded, the overloaded version are used.

```
#include <iostream>
#include <cstdlib>
#include <new>
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";
  }
};
```

```
// Global new
void *operator new(size_t size)
{
  void *p;
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}
// Global delete
void operator delete(void *p) {
  free(p);
}
int main()
{
  loc *p1, *p2;
  float *f;
  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for
p1.\n";
    return 1;
  }
```

```
  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for
p2.\n";
    return 1;
  }
  try {
    f = new float; // uses overloaded
                   //new, too
  } catch (bad_alloc xa) {
    cout << "Allocation error for f.\n";
    return 1;;
  }
  *f = 10.10F;
  cout << *f << "\n";
  p1->show();
  p2->show();
  delete p1;
  delete p2;
  delete f;
  return 0;
}
```

overloading new & delete for arrays

if you want to be able to allocate arrays of objects using your own allocation system, you will need to overload *new* & *delete* a second time.

```
//allocate an array of objects
void * operator new[] (size_t, size)
{
        /* 1. perform allocation throw bad-allocation on failure
           2. constructor for each element called automatically
           3. return pointer-to-memory; */
}
// delete an array of objects
void operator delete[] (void*p)
{
        /* 1. Free memory pointed to by p
           2. Destruction for each element called automatically */
}
```

Note:- When allocating an array, the constructor function for each object in the array is automatically called. when freeing an array, each objects destructor is automatically called. you do not have to provide explicit code to accomplish these actions.

overloading new & delete for arrays

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
  int longitude, latitude;
public:
  loc() {longitude = latitude = 0;}
  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);
  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);
if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}
// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
  cout << "In overloaded delete.\n";
  free(p);
}
// new overloaded for loc arrays.
void *loc::operator new[](size_t size)
{
  void *p;
  cout << "Using overload new[].\n";
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}
// delete overloaded for loc arrays.
void loc::operator delete[](void *p)
{
  cout << "Freeing array using overloaded
delete[]\n";
  free(p);
}
int main()
{
  loc *p1, *p2;
  int i;
  try {
    p1 = new loc (10, 20);
          // allocate an object
  } catch (bad_alloc xa) {
    cout << "Allocation
            error for p1.\n";
    return 1;;
  }
  try {
    p2 = new loc [10];
          // allocate an array
  } catch (bad_alloc xa) {
    cout << "Allocation
            error for p2.\n";
    return 1;;
  }
  p1->show();

  for(i=0; i<10; i++)
    p2[i].show();
  delete p1; // free an
                //object
  delete [] p2; // free an
                  //array
  return 0;
}
```

Type conversion

int m;
float x = 3.14159;
m = x; // automatic type conversion take place
cout<<m; // 3 will be displayed

Three situations where data conversion is needed to be taken care by the programmer

→How the conversion from built-in type to class type will take place { constructor with single argument}
→How the conversion from class type to built-in type take place { overloading casting function/ converts an operator}
→how the conversion from one class type to another class type.
Will take place -> {constructor in destination class / conversion operation in source class}

Basic to class type

```
Class string {
                char *p;
                int len;
                string (char *a);
}
string :: string (char *a) {
                length = strlen (a);
                p = new char [length + 1];
                strcpy(p,a);
}
char *name1 =  “xyz”;
char *name2 =  “abc”;
string s1 =  name1;  // an string s1 = string (name1);
                        // constructor will be called
class time
{
                int hrs;
                int mins;
                public:
                time (int t)
                {
                                hrs = t/60;    // t in minutes
                                mins = t%60;
                }
};
```

```
main()
{
   int duration = 85;//duration in minutes
    time T1 = duration; //  int to class type
    time T2;
    T2 = duration // valid , first it will look for
                        // appropriately overloaded
                       // assignment operator and if it
                      //  is not found it will call any
                      //  function (constructor or
                     //  conversion function) to
                     //  achieve the conversion
}
```

Class to basic type

Conversion function

```
Operator typename () {
                .
                .       }
```

```
const size = 3;
class vector {
        int v[size];
        operator double();
        }
//scalar magnitude of a vector is calculated as the square root of the sum of the squares of its components
vector :: operator double()   {
       double sum = 0;
       for (int i =0;i<size; i++)
                  sum = sum + v[i]* v[i];
                return sqrt(sum);
              }

double length = v1; // calls operator double ()
double length ;
length = v1; // calls operator double()
length = (double) v1; // calls operator double()
length = v1;      //  valid , first it will look for appropriately overloaded assignment operator and if it is not found it
                  //  will call any function (constructor or conversion function) to achive the conversion although
                  //   some books appreciate the syntax as given below  → static_cast<double>(v1);
```

→The casting operator (or conversion) function should satisfy the following conditions.

- → it must be a class member
- → it must not specify a return type (as by default it is the basic type to which the class is being converted)
- → it must not have any arguments (the invoking object is used as an argument)

one class to another class

obj x = obj y

destination class = source class

→two ways for achieving such conversion

- **→have a single argument constructor in destination class taking object of source class as argument**
- **→have a conversion operator/ function to destination class in the source class definition.**
- **→ example on next slide**

one class to another class

```
#include<iostream.h>
class invent1          {        //source class
                int code;
                int items;     //no. of files
                float price;   //case of each file
  public:
                invent1(int a, int b, float )
                {
                                ------
                }
                void putdata()
                {
                                cout<<"Code"<<code;
                }
                int getcode()    { return code;}
                int getitem()    {return item;}
                int getprice()   {return getprice;}

operator float(){
                        return(item*price);   }
 /* operator invent2() //invent2 = invent1  {
                invent2 temp;
                temp.code = code;
                temp.value = price*item;
                return temp;
                } */
```

```
class invent2 //destination class  {
public:
                int code;
                float value;
                invent2 (){
                }
                invent2(int x, float y) {
                }
void putdata(){
                    ---
                }

invent2(invent1 p) // conversion constructor requires
                        //public function in invent1 which
                        //can access private members of
                        //invent1
{          code = p.getcode();
           value = p.getitems () * p.getprice ();
}
};
main()   {
                invent1 s1(100,9,140);
                invent2 d1;
                float total-value;
                total-value = s1; // operator float()
                d1 = s1;  //invent2 (invent1) –constructor of invent2
                ----
}
```

Overloading some special operator

Keyword → explicit → single argument constructor with explicit keyword can not be used for implicit conversion

```
class ABC
{
        ----
        explicit ABC (float i)
        {
        }
}
int main()
{
        ABC x = 37.4F; //error
        ABC  y;
        y = 37.4F; // error
        ABC x(37.4F); // OK
}
```

Keyword → mutable → a mutable data member of a class can be modified even if the object of this class is declared as constant

```
class ABC  {
        int a;
        mutalble float b;
    public:
        ABC (int , float j);
        void seta(int a1) {
                a=a1;
        }
        void setb(float b1) {
                b=b1;
        }
}
int main()
{
        const ABC x(2,3.14F);
        x.seta(5); //error
        x.setb(7.14F);// allowed as b is mutable
}
```