WEEK 8
OPERATOR OVERLOADING

Dr. Yuzana Win

TOPICS COVERED

- Overloading Unary Operators
- Overloading Binary Operators
- Data Conversion

OPERATOR OVERLOADING

- Polymorphism is also achieved in C++ through operator overloading.

- When an operator is overloaded, it takes on an additional meaning relative to a certain class. However, it still retains all of its old meanings.

- **Example**:- In C++, it is possible to use **<<** and **>>** operators to perform console I/O operations. They can perform these extra operations because in the **<iostream> header,** these operators are overloaded.

RULES

- It is not recommended to change the basic meaning of an operator.
- It cannot be overloaded the way an operator works with the intrinsic C++ data Types.
- New operator cannot be invented that do not exist in C++ language.
- The precedence of operator cannot be changed.
- The following operators cannot be overloaded.

  . Class member operator
  .* Pointer to member operator
  :: Scope resolution operator
  ?: Conditional operator

CREATING A MEMBER OPERATOR FUNCTION

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
  // operations
}
```

- *ret-type* ➔ can be any valid type
- *arg-list* ➔ *empty*, when overloading a unary operator.

  → *one parameter, (object or primitive type)* when overloading a binary operator.

EXAMPLE: OVERLOADING 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 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;
}
```

EXAMPLE: OVERLOADING UNARY OPERATOR ++

```
#include <iostream>
using namespace std;
class Counter
{
    private:
        unsigned int count;
    public:
        Counter() { count = 0;}
        int get_count()
        {
            return count;
        }
        void operator ++ ()
        {
                ++count;
        }
};
```

```
int main()
{   Counter c1, c2;
    cout << “\nc1=”  <<
    c1.get_count(); cout << “\nc2=”
    << c2.get_count();
    ++c1;
    ++c2;
    cout << “\nc1=”  <<
    c1.get_count(); cout << “\nc2=”
    << c2.get_count();
    return 0;
}
```

Output ?

OVERLOADING OF BINARY ARITHMETIC OPERATOR

Complex operator+(int){}

- This function will be called by compiler when it sees an expression that has an object of the class type on the left side of the operator and an object of the parameter on the right.

- *Example*

    Complex obj(2,3);

    Complex obj1=**obj+5**;

OPERATOR OVERLOADING USING A **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.

EXAMPLE: OVERLOADING OPERATOR+() AS FRIEND

```
#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, loc
op2);
      loc operator-(loc op2);
      loc operator=(loc op2);
```

```
      loc operator++();
};
 loc operator+(loc op1, loc op2)
{
        loc temp;
        temp.longitude = op1.longitude +
                                   op2.longitude;
        temp.latitude = op1.latitude +
                                op2.latitude;
        return temp;
}
loc loc::operator-(loc op2)
{
   loc temp;
   temp.longitude = longitude -
                             op2.longitude;
   temp.latitude = latitude -
op2.latitude;
   return temp; }
```

EXAMPLE: OVERLOADING OPERATOR+() AS FRIEND

```
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this;
}

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

```
int main()
{
 loc ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();    //15  50
    loc obj3=ob1-ob2;
    obj3.show();   //10  20
    ob1=ob2;
    ob1.show();  // 5   30
     ++ob1;
     ob1.show();//6    31
     return 0;
}
```

OVERLOADING OF BINARY ARITHMETIC OPERATOR

Non-member operator function

- In case of member operator function always need object for which function is executing is on left side.
- Non-member operator function can be used to overload the operator with a different type on the left and different type on the right.
- *Example*

```
Complex obj1=100+obj;
Complex operator+(int n, Complex& dt)
```

OVERLOADING OF **BINARY ARITHMETIC OPERATOR**

Example

```
class Complex{
        friend Complex operator+(int n, Complex&);
        }
```

- We can make this nonmember function as <u>friend function</u>, so that it can have access to the private members of the class.
- Nonmember Operator Function can also be used to add two objects of class.

OVERLOADING OF << AND >> OPERATOR

```
#include <iostream>
using namespace std;
class A
{
  int a1,a2;
  public:
   friend istream& operator>>(istream&,A&);
   friend ostream& operator<<(ostream&,A&);
};
```

```
int main()
{ A obja;
   cout<<"Enter a1 & a2 for obj: ";
   cin>>obja;
   cout<<obja;
   return 0;
}
```

```
istream& operator >> (istream& is, A &obj)
{
        is>>obj.a1>>obj.a2;
        return is;
}
ostream& operator<<(ostream& os,A &obj)
{   os<<"a1="<<obj.a1<<"\na2= "<<obj.a2;
    return os;
}
```

Output:

```
Enter a1 & a2 for obj:
20 100
a1=20
a2=100
Press any key to continue . . .
```

OVERLOADING OF UNARY OPERATOR

Prefix increment

type operator++( ) {

    // body of prefix operator

}

Prefix decrement

type operator- -( ) {

    // body of prefix operator

}

Postfix increment

type operator++(int x) {

    // body of postfix operator

}

Postfix decrement

type operator- -(int x) {

    // body of postfix operator

}

- **int** arg is a dummy argument used to distinguish between prefix and postfix operator ;
- Depending on the notation, compiler calls appropriate function.

OVERLOADED ASSIGNMENT OPERATORS

- Example
```
Complex obj(2,3);
Complex newobj(7,20);
Newobj=obj;
```
- It is implemented as a member-by member copy.
- Disadvantage is overcome by Overloaded Assignment Operators,
```
void operator=(const Date&);
```
- Parameter represents source object and operator function runs as a member function of destination object.

OVERLOADING THE SHORTHAND OPERATORS

- You can overload any of C++'s "shorthand" operators, such as **+=**, **-=**, and the like.

- For example, the following function overloads **+=** relative to loc:

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```

FRIEND FUNCTIONS ADD FLEXIBILITY

- Using **member operator** functions,
  - ✓ obj = obj + 100; // valid
  - ✗ obj= 100 + obj; // invalid

- Using **friend** functions,
  - ✓ obj = obj + 100; // valid
  - ✓ obj= 100 + obj; // valid

EXAMPLE: USING FRIEND FUNCTION

```
#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); };
```

```
loc operator+(loc op1, int op2)
{
   loc temp;
   temp.longitude = op1.longitude +
op2;
   temp.latitude = op1.latitude + op2;
   return temp;
}
loc operator+(int op1, loc op2)
{
    loc temp;
    temp.longitude = op1 +
op2.longitude;
    temp.latitude = op1 + op2.latitude;
    return temp;
}
```

EXAMPLE: USING FRIEND FUNCTION

```
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 AND DELETE**

- It is possible to overload new and delete.
- You might choose to do this if you want **to use some special allocation method.**
- For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted.
- Whatever the reason, it is a very simple matter to overload these operators.

SYNTAX

```
// 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. */
}
```

OVERLOADING NEW AND DELETE FOR ARRAYS

```
// 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. */
}
```

OVERLOADING [ ]

```
type class-name::operator[ ](int i)
{
        // . . .
}
```

- The operator[ ]() function is typically used to provide array subscripting, and as such, an integer value is generally used.

- Given an object called O, the expression

  O[3]

  translates into this call to the operator[ ]() function:

  O.operator[](3)

EXAMPLE: OVERLOADING [ ]

```
#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;
}

/* int& operator[](int i) can be used
on both left and right of an
assignment statements.
e.g.,
     cout << ob[1];
     ob[1] = 25; // [] on left of =
     cout << ob[1];
*/
```

Output:

```
2
25
Press any key to continue . . .
```

OVERLOADING ( )

- Given the overloaded operator function declaration

  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");

OVERLOADING ->

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

- Its general usage is:

  object->element;

- 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.

EXAMPLE:OVERLOADING ->

```
#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
Press any key to continue . . .
```

OVERLOADING THE COMMA OPERATOR

- The comma is a **binary operator**, and like all overloaded operators, you can make an overloaded comma perform any operation you want.

- However, if you want the overloaded comma to perform in a fashion similar to its normal operation, then your version must discard the values of all operands except the rightmost.

- The **rightmost value becomes the result of the comma operation**. This is the way the comma works by default in C++.

KEY POINTS TO REMEMBER

- Friend functions can not be used to overload these operators ==,(),[] and ->.
- Unary operator, overloaded by means of member function take no explicit arguments
- Binary operators overloaded through a member function take one explicit argument and those that are overloaded through a friend function take two arguments.
- Binary operators such as +,-,* and / must explicitly return a value.
- Operator functions can not have default value.

COPY CONSTRUCTOR

- Copy constructor initialize the values of new, instantiated object from an existing object of the same class.
- General declaration is:

  Complex(const Complex&);
- Copy constructor takes copy of an object of the class as an argument.
- Copy constructor returns an object of same class.
- A copy constructor is executed when we initialize a new object of the class with the existing object of the same class.
- Copy constructor can be called by using syntax

  Complex newobj=obj;
  Complex newobj(obj);
- If no Copy Constructor is provided, compiler provides one, by which one object can created by making bit by bit copy of other object.

COPY CONSTRUCTOR AND ASSIGNMENT OPERATOR

- In C++, Assignment operator defines as **shallow copy** and Copy constructor defines as **deep copy**.
- Assignment operator assigns the value of an existing object to another existing object.
- Copy constructor creates a new object and initialize it with the content of the existing object.
- Compiler distinguish the two by using overloaded assignment operator for assignment and copy constructor for initialization.

EXAMPLE OF COPY CONSTRUCTOR AND ASSIGNMENT OPERATOR

```
#include <iostream>
using namespace std;
class A
{   int i;
   public:
    A()
    { i=10;}
    A (const A &aa)
     { i=aa.i; }
    void display()
     { cout<<i; }
};
```

```
int main()
{ A a;
  A a1(a); // calling copy constructor
  A a2=a1; // calling copy constructor
  a.display();
  a1.display();
  a2.display();
  A a3;
  a3=a;    // assignment
  a3.display();
}
```

Output
10
10
10
10

Remark:
- copy constructor for initialization
- assignment operator is only copy from existing object's members to another object's members.

REFERENCE VARIABLE

Why we use reference variable in copy constructor?

- Reference is an alias for another variable.

  Example,

  int actualint;
  int& otherint=actualint;

- Reference variables must be initialized when it is declared (except in few cases).
  - When it is declared with extern.
  - When it is declared as argument in function declaration.
  - When it is a data member of the class.

- When it is initialized, the value can not be changed for lifetime of the reference.
- There is no such thing as null reference.
- Reference variable and actual variable will have same memory address.
- We can also return reference from a function.

ADVANTAGES OF REFERENCE

Why we use reference variable in copy constructor?

- To reduce complex notation.
- Eliminates overhead of copying large data structures(because of pass by reference).
- It eliminates * operator.
- We can not modify the reference so eliminate much of a trouble with the pointers.

Const Reference Parameters

- When we want to prevent a function from being able to modify its caller’s referenced variable.

DATA CONVERSION

Conversion Function

Source

1. write conversion function
2. start with operator ()
3. no need to return but write return type

Destination

1. write Constructor function (argument)
2. same class name
3. no need to return

CONVERSIONS BETWEEN OBJECTS AND BASIC TYPES

Conversions: Distance to meters, meters to Distance

```
#include <iostream>
using namespace std;
const float MTF = 3.280833;
class Distance
{ private:
     int feet;
      float inches;
   public:
      Distance ()
      { feet = 0;
         inches = 0.0;}
      Distance (float meters)
      { float f = MTF * meters;
        feet = int (f);
        inches = 12 * ( f –
     feet); }
```

```
Distance (int ft, float in)
{    feet = ft; inches =
     in; }
void getdist( )
{  cout << “Enter feet:”;
    cin    >> feet;
    cout << “\n Enter
     inches:”;
   cin  >> inches; }
void showdist()
 {cout << feet << "\'-" <<
     inches << '\"'; }
operator float()
{ float f = inches / 12;
          f += float (feet);
    return f / MTF; }};
```

```
int main( )
{    Distance dist1 = 2.35
       float m;
      cout << “\n dist1=”;
     dist1.showdist();
     m = float (dist1);
     cout << “\n dist1 =”
      << m << “meters\n”;
    Distance dist2(5,
      10.25);
    m = dist2;
    cout << “\n dist2 =” <
      m << “meters\n”;
  return 0;
}
```

CONVERSIONS BETWEEN OBJECTS AND BASIC TYPES

Output

```
 dist1=7'-8.51949"
 dist1 =2.35meters

 dist2 =1.78435meters
Press any key to continue . . . _
```

ASSIGNMENT

1. overload =, ==, +, ++, --, <<, >> and [] operators

2. Create a class *Distance* which includes two data members: *feet* and *inches*, two constructors: default constructor and parameter constructor and three member functions: *getdist*, *showdist* and add an *overloaded -operator* that subtracts two distances. Create three objects in the main: two objects *dist1* and *dist3* by no argument constructor and another object *dist2* by parameter constructor. Update the *dist1* value by calling member function *getdist*. Then subtract two objects *dist1* and *dist2* and assign it to *dist3*. Show the data in *dist1*, *dist2* and *dist3* by calling *showdist* function.