Operator Overloading

Compile Time Polymorphism

- Compile time polymorphism comes in two flavors.

- Function Overloading
  - Function name with a variable number and type of arguments.
  - void fun(int, int, double), void fun(float, float, int), etc..

- Operator Overloading
  - Operator used to operate on user defined types similar to built-in types of the language.

  - + , - , *, / operators that operate on user defined types, i.e., on objects of the classes.

Basics of Operator Overloading

- Example:
- a = b + c; // Assume a, b, c as built in types
- a = b + c // all are integers
- a = b + c // all are doubles
- a = b + c // b is int, c is double and a is double
- a = b + c // a, b and c are objects of a class.

- Can we able to add/subtract objects/user-defined types?
- What do you we mean by adding/subtracting them?
- For ex., If we try to add them, then will compiler complain?

Basics of Operator Overloading Cont..

- a = b + c // a, b and c are objects of a class.
- This is can be done in C++ using a concept of Operator Overloading.
- A function called an ***operator function*** will be used by compiler implicitly to accomplish the desired operation.
- For. ex. the addiion b + c will be treated as: b.**operator +** (c)
  // first argument (b) is used implicitly.
- Since we know which operators take what operands at compile time, the said polymorphism is called compile time polymorphism.

Basics of Operator Overloading Cont..

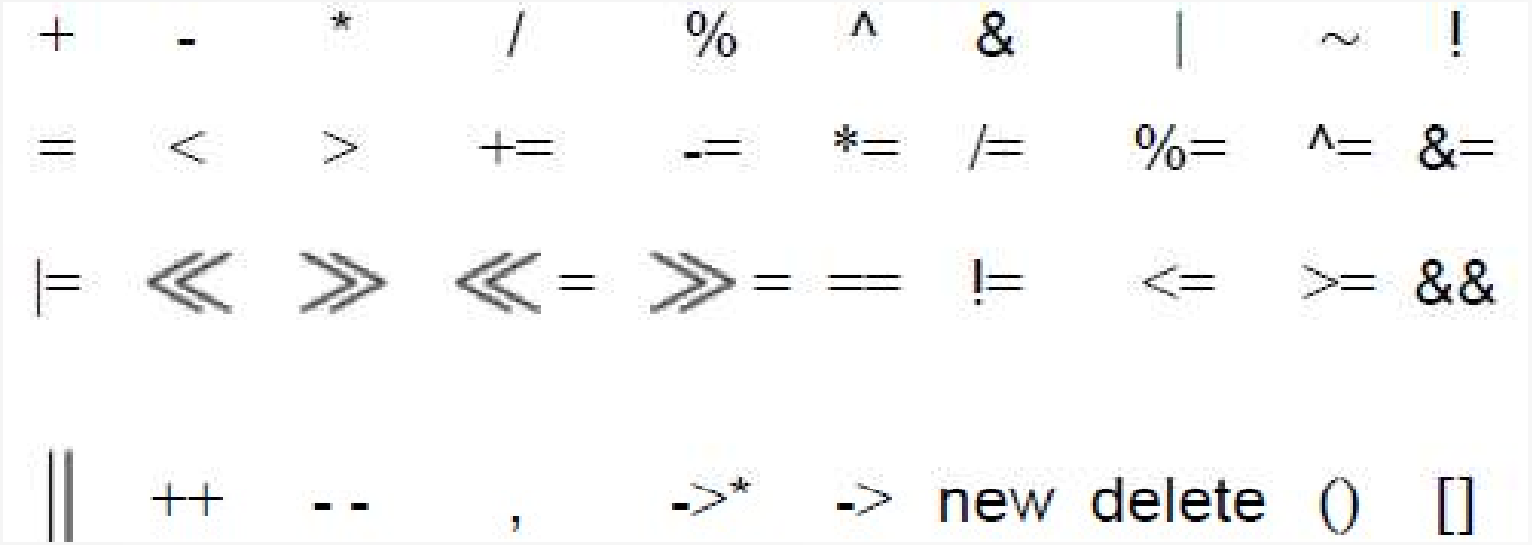

Fig. 1. Operators that can be overloaded

'.', '.*', '::' and '?:'

Fig. 2. Operators that can not be overloaded

Basics of Operator Overloading Cont..

- **Operator Rules**
- While performing the operations with the operators, the meaning of the operator should not be changed, e.g., ‘+’ should be used for addition only, ‘-’ should be used for subtraction only, and so on.

- The number of operands (arguments to an operator function) should also remain the same, e.g., binary ‘+’ takes two operands (arguments), unary - takes one operand (argument), etc.

- The precedence and associativity should also remain same.

Precedence and Associativity

- Two operator characteristics determine how operands group with operators
  - Precedence: the priority for grouping different types of operators with their operands.
  - Associativity:the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.
  - c = a + b * d; The operation assigned to '*' will precede the execution of '+'.
  - c = b = a; //will always be equivalent to c = (b = a) as the assignment operator is right associative.
  - c + b - a; // + and - have same precedence and both are having same associativity. Both of them are left-associative, so addition is done before subtraction.

Binary Operator Overloading

- Two versions to overload operators.
  - Using a member function of the class
  - Using a non-member function (friend function)

- For overloading a binary operator, an operator function with two arguments must be called of which at least one has to be of a user-defined type.
- The prototype of such a function would be:

  Type operator@(Type1, Type2);

- The type of the returned value is arbitrary, i.e., it can be any return type or even void.

Binary Operator Overloading Cont..

- The function can be a friend of our class if we would like to have a direct access to private and protected members of the class.
- The operator member function that overloads a binary operator needs will have only one parameter.
- This parameter will receive the object that is on the right side of the operator.
- The object on the left side is the object that generates the call to the "**operator function"** and is passed implicitly by using "this" pointer.

Binary Operator Overloading Cont..

```
erload the + relative to
d class
s coord

 y; // coordinate values
c :
d () { x=0; y=0; };
d (int i, int j) { x=i; y=j; }
get_xy (int &i, int &j)
 j=y; }
d operator +( coord ob2);
```

```
// Overload + relative to
coord class .
coord coord :: operator
+( coord ob2)
{
coord temp ;
temp .x = x + ob2 .x;
temp .y = y + ob2 .y;
return temp ;
}
```

```
int main ()
{
coord o1 (10 , 10) ,
o2 (5, 3) , o3;
int x, y;
o3 = o1 + o2; // add
two objects -
this calls operator +
o3. get_xy (x, y);
cout << "(o1+o2) X: "
<< x << ", Y: " << y <<
"\n";
return 0;
}
```

Binary Operator Overloading Cont..

```
class coord
{
int x, y; // coordinate values
public :
coord () {x=0; y=0;};
coord (int i, int j) {x=i; y=j;}
void get_xy (int &i, int &j) {i=x;
=y;}
coord operator +(coord ob2);
coord operator -(coord ob2);
coord operator =(coord ob2);
};
```

```
//Overload + relative to
coord class.
coord coord :: operator +
(coord ob2)
{
coord temp ;
temp .x = x + ob2 .x;
temp .y = y + ob2 .y;
return temp;
}
```

```
// Overload - relative to
coord class .
coord coord :: operator -
(coord ob2)
{
coord temp ;
temp .x = x - ob2 .x;
temp .y = y - ob2 .y;
return temp;
}
```

Binary Operator Overloading Cont..

```
 Overload = relative to coord
ass .
oord coord :: operator = (coord
b2)

 = ob2.x;
 = ob2.y;
/ return the object that is
ssigned
eturn * this ;
```

```
int main ()
{
coord o1 (10 , 10) , o2 (5, 3) , o3;
int x, y;
o3 = o1 + o2; // add two objects - this calls operator +
o3. get_xy (x, y);
cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";
o3 = o1 - o2; // subtract two objects
o3. get_xy (x, y);
cout << "(o1 -o2) X: " << x << ", Y: " << y << "\n";
//o3 = o1; // assign an object
o3.operator = (o1);
o3. get_xy (x, y);
cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";
return 0;
}
```

OVERLOADING THE RELATIONAL AND LOGICAL OPERATORS

- Overloading the relational and logical operators will return will return an integer that indicates either true or false.

- This will lead to return a true/false value

- It will also let the operators to be integrated into larger relational and logical expressions that will have other types of data.

Example

```
class coord
{
int x, y; // coordinate values
public :
coord () { x=0; y=0; };
coord (int i, int j) { x=i; y=j; }
void get_xy (int &i, int &j)
{ i=x; j=y; }
int operator ==( coord ob2);
int operator &&( coord ob2);
};
```

```
// Overload the ==
operator for coord .
int coord :: operator
==( coord ob2)
{
return x== ob2.x &&
y== ob2.y;
}
// Overload the &&
operator for coord .
int coord :: operator
&&( coord ob2)
{
return (x && ob2.x) &&
(y && ob2.y);
}
```

```
int main () {
coord o1 (10 , 10) , o2 (5, 3) ,
 o3 (10 , 10) , o4 (0, 0);
if(o1 == o2)
cout << "o1 same as o2\n";
else
cout << "o1 and o2 differs \n";
if(o1 == o3)
cout << "o1 same as o3\n";
else
cout << "o1 and o3 differ \n";
if(o1 && o2)
cout << "o1 && o2 is true \n";
else
cout << "o1 && o2 is false \n";
if(o1 && o4)
cout << "o1 && o4 is true \n";
else
cout << "o1 && o4 is false \n";
return 0;
}
```

UNARY OPERATOR OVERLOADING

- It is similar to binary operator overloading.
- Using a member function requires no parameters.
- The only operand generates the call to the operator function.
- This pointer holds the address of this operand (object).

Prefix Unary Operator

```
s coord

 y; // coordinate values
c :
d () { x=0; y=0; };
d (int i, int j) { x=i; y=j; }
get_xy (int &i, int &j)
 j=y; }
d operator ++() ;
```

```
// Overload ++
for coord .
coord coord ::
operator ++()
{
++x;
++y;
return * this ;
}
```

```
int main ()
{
coord o1 (10 , 10), o2;
int x, y;
// increment an object
++o1; o1. get_xy (x, y);
cout << "(++ o1) X: " << x << ",
Y: " << y << "\n";
// increment an object
o2 = ++ o1;
o2. get_xy (x, y);
cout << "(o2) X: " << x << ", Y:
<< y << "\n";
return 0;
}
```

Prefix and Postfix Unary Operators

```
coord

y; // coordinate values
 :
 () { x=0; y=0; };
 (int i, int j) { x=i; y=j; }
et_xy (int &i, int &j)
=y;}
 operator ++() ;
 operator ++(int notused) ;
```

```
// Overload ++ for coord
.
coord coord :: operator
++(){
++x;
++y;
return * this ;
}
// Overload ++ for coord
.
coord coord :: operator
++(int notused) {
x++;
y++;
return * this ;
}
```

```
int main ()
{
coord o1 (10 , 10), o2;
int x, y;
++o1; // increment an object
o1. get_xy (x, y);
cout << "(++ o1) X: " << x << ", Y:
<< "\n";
o2 = ++ o1; // increment an object
o2. get_xy (x, y);
cout << "(o2) X: " << x << ", Y: " <<
"\n";
o2++;
o2. get_xy (x, y);
cout << "(o2) X: " << x << ", Y: " <<
"\n";
return 0;
}
```

Unary and Binary '-' Operators

```
rload the - relative to coord

de <iostream >
namespace std;
coord {
; // coordinate values
:
() { x=0; y=0; };
(int i, int j) { x=i; y=j; }
et_xy (int &i, int &j) { i=x; j=y; }
ry minus
operator -( coord ob2);
y minus
operator -();
```

```
// Overload - relative to
coord class .
coord coord :: operator -
(coord ob2) {
coord temp ;
temp .x = x - ob2 .x;
temp .y = y - ob2 .y;
return temp ;
}
// Overload unary - relative
to coord class .
coord coord :: operator -()
{
x = -x;
y = -y;
return * this ;
}
```

```
int main ()
{
coord o1 (10 , 10) , o2 (5, 7);
int x, y;
o1 = o1 - o2; // subtraction
o1. get_xy (x, y);
cout << "(o1 -o2) X: " << x << ", Y: " <
y << "\n";
o1 = -o1; // negation
o1. get_xy (x, y);
cout << "(-o1) X: " << x << ", Y: " << y
<< "\n";
return 0;
}
```

FRIEND OPERATOR FUNCTIONS

- Operator function can be a Friend function.
- It does not take /pass a this pointer.

- The operator function must be passed with all the required operands explicitly.

- Binary operator function - 2 arguments.
- Unary operator function - 1 argument.

Binary '+' Operator Overloading with Friend

```
coord

; // coordinate values
:
() { x=0; y=0; };
(int i, int j) { x=i; y=j; }
et_xy (int &i, int &j) { i=x; j=y; }
coord operator +( coord ob1 ,
ob2 );
```

```
// Overload + using a friend
.
coord operator +( coord
ob1 , coord ob2)
{
coord temp ;
temp .x = ob1 .x + ob2 .x;
temp .y = ob1 .y + ob2 .y;
return temp ;
}
```

```
int main ()
{
coord o1 (10 , 10) , o2 (5, 3) , o3;
int x, y;
o3 = o1 + o2; // add two objects - this
calls operator +()
o3. get_xy (x, y);
cout << "(o1+o2) X: " << x << ", Y: " <
y << "\n";
return 0;
}
```

FRIEND OPERATOR FUNCTIONS

- In case of operations between built in type and user defined type, using friend function, the following is legal.

- class coord { .. } ;
- coord coord:: operator+ (coord, int) { .. }
- coord ob1, ob2;
- ob1 = 5 + ob2; // Note, int on left-hand side is ok

FRIEND OPERATOR FUNCTIONS Cont..

- Since the member function has an access to this pointer, the modifications can be done using this pointer in the overloaded function.

- For unary operator like ++, friend function must be passed with the reference parameter, then only the changes will be reflected in the object that is passed by reference at the time of calling.

Unary '++' Operator Overloading with Friend

```
rload the ++ using a friend .
de <iostream >
namespace std;
coord

; // coordinate values
:
() { x=0; y=0; };
(int i, int j) { x=i; y=j; }
et_xy (int &i, int &j) { i=x; j=y; }
coord operator ++( coord
```

```
// Overload ++ using a
friend .
coord operator ++( coord
&ob)
// use reference parameter
{
ob.x++;
ob.y++;
return ob;
}
```

```
int main ()
{
coord o1 (10 , 10);
int x, y;
++ o1; // o1 is passed by reference
o1. get_xy (x, y);
cout << "(++ o1) X: " << x << ", Y: "
<< y << "\n";
return 0;
}
```

References

- C++: The Complete Reference, 4th Edition by Herbert Schildt , McGraw-Hill
- Teach Yourself C++ 3rd Edition by Herbert Schildt,
- The C+ + Programming Language, Third Edition by Bjarne Stroustrup, Addison Wesley