Operator Functions as Class Members vs. as friend Functions

- ***Operator functions*** can be member or non-member functions…
- The following operators, must use a ***member function***:

`( )`	Function call
`[ ]`	Array subscript
`->`	Member Selection
`= *= /= %= += -= <<= >>= &= |= ^=`	Assignment operators

- Operator functions as ***member functions***:
  - Leftmost (or only) operand must be an object (or reference to an object) of the class…
- Operator functions as ***non-member functions***:
  - If left operand of a different type or an object of a different class, operator function must be a non-member function…
  - Must be friends if needs to access private or protected members… (Otherwise Set /Get function are needed)
  - Enable the operator to be commutative… (**a + b** or **b + a**)


```
#include <iostream.h>
class Point {
   // Friend function declarations
   friend Point operator+( int val, Point& pt);
   public:
      Point(int x, int y) {valX =x; valY =y; }
      void Print() {
         cout << valX << ", " << valY << endl;
      }
      Point operator+( Point& pt ) {
         pt.valX = pt.valX + valX;
         pt.valY = pt.valY + valY;
         return pt;
      }
      Point operator+(int val ) {
          valX += val;
          valY += val;
          return *this;
      }
   private:
      int valX;
      int valY;
};
```



```
Point operator+( int val, Point& pt ) {
    pt.valX += val;
    pt.valY += val;
    return pt;
}
```

```
void main() {
    Point pt1(10,20), pt2(2,5);
    pt1.Print();
    pt1 = pt2 + pt1;     // Point + Point
    pt1.Print();
    pt1 = 3 + pt1;       // int + Point
    pt1.Print();
    pt1 = pt1 + 5 ;      // Point + int
    pt1.Print();
}
```


- Operator functions can be **non-member / non-friend** functions (Poor performance)…

```
#include <iostream.h>

class A {
   public:
      A (int a) {valA = a;}
      void setA(int a) {valA = a;}
      int getA(void) {return(valA);}
   private:
      int valA;
};
void operator += ( int i, A& aRef ) {
   aRef.setA(aRef.getA() + i);
}
void main() {
   A a1(10), &aRef = a1;
   operator +=(5,a1);
   cout << a1.getA();
}
```

Overloading Unary Operators

- Overloaded as a member functions with no argument… They must be non-static so they can access the non-static data of the class…
- Overloaded as a non-member functions with one argument…(Object or reference to object of that class)

```
#include <iostream.h>
class Number {
   friend int operator- ( Number &) ;
   public:
      Number (int valN ) {nbr = valN;}
      bool operator!() const { return(nbr <0);}
   private:
      int nbr;
};
int operator-(Number &nRef)  {
   return(-nRef.nbr);
}
void main() {
   Number n1(-10);
   if (!n1)
      cout << -n1;
}
```


Overloading Binary Operators

- Overloaded as a member functions with one argument… They must be non-static so they can access the non-static data of the class…
- Overloaded as a non-member functions with two argument…(One of witch must be an Object or a reference to object of that class)

```
#include <iostream.h>
class Number {
   friend void operator -= ( Number &, int);
   public:
      Number (int n) {valN = n;}
      void operator += ( int i ) {
         valN = valN + i;
      }
      void print() {cout <<valN <<endl;}
   private:
      int valN;
};
void operator -= ( Number &nRef, int i) {
   nRef.valN = nRef.valN - i;
}
void main() {
   Number n1(10);
   n1.print();
   n1 -=10;
   n1.print();
   n1 +=5;
   n1.print();
}
```


Overloaded *Assignment Operator* and *Copy Constructor*

- The default method of copying objects is not the best way when a data member pointer points to dynamic data…

```
class person {
   public:
      person(char nm[]="\0") {
         name = new char[strlen(nm)+1];
         strcpy(name,nm);
      }
      ~person() {
         delete [] name;
      }
      void print() const{
         cout << name << endl;
      }
   const person &operator=(const person &);
   private:
      char *name;
};
```

```
void main() {
   person agent("Bond...007");
   person *ptr;
   ptr = new person("James Bond");
   agent.print();
   ptr->print();
   agent = *ptr;
   agent.print();
   ptr->print();
   delete ptr;
   agent.print();
}
```



```
// Overloaded assignment operator
const person & person::operator=(const person &fromObj) {
   if ( &fromObj != this ) {   // check for self-assignment
      if ( strlen(name) != strlen(fromObj.name)) {
         delete [] name;  // Different sizes, deallocate original
         name = new char[strlen(fromObj.name)+1];
      }
      strcpy(name,fromObj.name);
   }
   return *this;        // Cascading  x = y = z;
}
```

```
class person {
   public:
      person(char nm[]="\0") {
         name = new char[strlen(nm)+1];
         strcpy(name,nm);
      }
      person(const person &prs) {// Copy Constructor
         name = new char[strlen(prs.name)+1];
         strcpy(name,prs.name);
      }
      ~person() {
         delete [] name;
      }
      void print() const{
         cout << name << endl;
      }
   private:
      char *name;
};
```

```
void main() {
   person *ptr = new person("John");
   person p1=*ptr;
   person p2=p1;
   person p3(p1);
   p1.print();
   p2.print();
   p3.print();
   delete ptr;
   p1.print();
   p2.print();
   p3.print();
}
```



```
class person {
   public:
      person(char nm[]="\0") {
         name = new char[strlen(nm)+1];
         strcpy(name,nm);
      }
      person(const person &prs) {// Copy Constructor
         name = new char[strlen(prs.name)+1];
         strcpy(name,prs.name);
      }
      ~person() {
         delete [] name;
      }
      void print() const{
         cout << name << endl;
      }
   private:
      char *name;
};
```

```
void iAm(person p) {
   p.print();
}
```

```
void main() {
   person p1("Joe");
   p1.print();
   iAm(p1);
   p1.print();
}
```



```
class person {
   public:
      person(char nm[]="\0") {
         name = new char[strlen(nm)+1];
         strcpy(name,nm);
      }
      person(const person &prs) {// Copy Constructor
         name = new char[strlen(prs.name)+1];
         strcpy(name,prs.name);
      }
      ~person() {
         delete [] name;
      }
      void print() const{
         cout << name << endl;
      }
   const person &operator=(const person &);
   private:
      char *name;
};
```

```
person whoAreYou() {
   person me("Gates");
   return me;
}
```

```
void main() {
   person p1("Bill");
   p1.print();
   p1 = whoAreYou();
   p1.print();
}
```



```
// Overloaded assignment operator
const person & person::operator=(const person &fromObj) {
   if ( &fromObj != this ) {   // check for self-assignment
      if ( strlen(name) != strlen(fromObj.name)) {
         delete [] name;  // Different sizes, deallocate original
         name = new char[strlen(fromObj.name)+1];
      }
      strcpy(name,fromObj.name);
   }
   return *this;        // Cascading  x = y = z;
}
```

Note:

- Copy constructor is called automatically…
- Like the constructor, it has no return type….

Converting between types

- The complier knows how to perform certain conversion among built-in types …

```
#include <iostream.h>
void fn(int val) {
   cout << val << endl;
}
void main() {
   float f1;
   int i1;
   f1=1.5;
   i1=f1;
   cout << f1 << endl;
   cout << i1 << endl;
   fn(f1);
}
```

But what about user-defined types…

- The compiler cannot know how to convert among user define types and built-in types…

  Conversion constructor…(Cast operator)


```
#include <iostream.h>
class A {
   public:
      A(float f1) { fVal = f1; }
      operator float () const { return fVal;}
      operator int() const { return fVal+0.5;}
   private:
      float fVal;
};
void fn(int val) {
   cout << val << endl;
}
void main() {
   float d1;
   A a1(1.5);
   d1 = a1;
   cout << d1 << endl;
   fn(a1);
}
```

- ***Conversion constructor*** does not specify a return type… The return type is the type in which the object its been converted… Must be non-static member function… It cannot be a friend function…


```
#include <iostream.h>
class A {
   public:
      A(float f1=0.0) { fVal = f1; }
      void setA(float f1) { fVal= f1;  }
   private:
      float fVal;
};

class B {
   public:
      B() { iVal = 10; }

      operator A() {
         A tmpA;
         if (iVal > 0)
            tmpA.setA(iVal);
         else
            tmpA.setA(-iVal);

         return tmpA;
      }

   private:
      int iVal;
};


void main() {
   A a1(1.5);
   B b1;

   a1 = b1;
}
```


Overloading ++ and - - operators

- The ***prefix*** and ***postfix*** versions of the ***increment*** and ***decrement*** operators can be separately overloaded…
- Each overloaded operator must have its distinct signature… Thus, the following rule is observed:
  - The ***prefix*** form of the operator is declared exactly the same way as any other unary operator…
  - The ***postfix*** form accepts an additional *dummy argument* of type *int*…

```
#include <iostream.h>
class A {
  friend ostream &operator<<( ostream &,
                              const A & );
   public:
      A(int x) { iVal = x;}
      A &operator ++() {
         iVal +=1;
         return *this;
      }
      A operator ++(int x) {
         A tmpA = *this;
         iVal +=1;
         return tmpA;
      }
      void print() {cout << iVal << endl;}
   private:
      int iVal;
};
```

Dummy argument to distinguish between prefix and postfix signatures…

```
ostream &operator<<(ostream &out,const A &aRef ){
    out << aRef.iVal;
    return out;
}

void main() {
   A a1(10);

   a1++;
   a1.print();
   cout << a1++ <<endl;
   a1.print();

   ++a1;
   a1.print();
   cout << ++a1 <<endl;
   a1.print();

}
```


- The ***prefix*** and ***postfix*** versions of the ***increment*** and ***decrement*** operators can be implemented as non-member function…

```
#include <iostream.h>
class A {
  friend ostream &operator<<( ostream &,
                              const A & );
     friend A &operator++(A &);
     friend A operator++(A &, int);

     A(int x) { iVal = x;}
     void print() {
        cout << iVal << endl;
     }

   private:
      int iVal;
};

A &operator ++(A &aRef) {
   aRef.iVal +=1;
   return aRef;
}
A operator ++(A &aRef,int xyz) {
   A tmpA = aRef;
   aRef.iVal +=1;
   return tmpA;
}
```

Dummy argument to distinguish between prefix and postfix signatures…


```
ostream &operator<<(ostream &out,const A &aRef ){
    out << aRef.iVal;
    return out;
}

void main() {
   A a1(10);

   a1++;
   a1.print();
   cout << a1++ <<endl;
   a1.print();

   ++a1;
   a1.print();
   cout << ++a1 <<endl;
   a1.print();

}
```

Summary on how the different operator functions must be declared.

Expression	Operator (@)	**Function member**	**Friend function**
@a	**+ - * & ! ~ ++ --**	`A::operator@()`	`operator@(A)`
a@	**++ --**	`A::operator@(int)`	`operator@(A, int)`
a@b or b@a	**+ - * / % ^ & | < > == != <= >= << >> && || ,**	**Left-side operand is an object of class A (a@b)**	**Left-side operand is not an object of class A (b@a)**
		`A::operator@(B)`	`operator@(B, A)`
a@b	**= += -= *= /= %= ^= &= |= <<= >>= [ ]**	`A::operator@(B)`	-
a(b, c...)	**()**	`A::operator()(B, C...)`	-
a->b	**->**	`A::operator->()`	-

- Where **@** is the overloaded operator, **a** is an object of class **A**, **b** is an object of class **B** and **c** is an object of class **C**.