Object-Oriented Programming

Lesson 9

Operator overloading
Operator= in complex objects
Automatic type conversion

Postfix and prefix operator overloading

The overloaded ++ and -- operators present a dilemma because you want to be able to call different functions depending on whether they appear before (prefix) or after (postfix) the object they're acting upon.

That is, the compiler differentiates between the two forms by making calls to different overloaded functions.

These operators can be implemented as either class member functions or global (friend) functions.

Postfix and prefix operator overloading

```
class X          {
   //..............
public:
   X& operator++();                    // prefix
   const X operator++(int);            // postfix
};
```

For the prefix version the compiler generates the following call:

```
x1.operator++()
```

For the postfix version the compiler passes a dummy constant value for the **int** argument (which is never given an identifier because the value is never used) to generate the different signature:

```
x1.operator++(int)
```

Postfix and prefix operator overloading

```
class X
{       int a;
public:
        X(int aa) { a=aa; }
        X& operator++();            //prefix
        const X operator++(int);    //postfix
};

X& X::operator++() //prefix
{
        a++;
        return *this;
}

const X X::operator++(int) //postfix
{
        X temp = *this;
        a++;
        return temp;
}
```

```
int main ()
{
  X x1(1), x2(2);
  x1 = x2++; // x1=2, x2=3
  x1 = ++x2; // x1=4, x2=4
  return 0;
}
```

Operator= in derived classes

```
class base
{  //…
public:
   base& operator= (const base& r);
};
```

```
class derived : public derived
{  //…
public:
   derived& operator= (const derived& r);
};
```

```
derived& derived::operator = (const derived& r)
{
   if (&r != this)
   {
     base::operator =(r);
     //…
   }
   return *this;
}
```

When implementing the operator= in the derived class, you can call the base-class operator= !

Operator= in derived classes

The compiler will automatically create a **type::operator=(type)** if you don't make one.

The behaviour of this operator mimics that of the automatically created copy-constructor: if the class contains objects (or is inherited from another class), the **operator=** for those objects is called recursively. This is called memberwise assignment. Afterwards bitwise assignment of the remaining data members is done.

Operator= and composition

```
class A
{  //…
public:
   //…
   A& operator= (const A& r);
};
```

Class B includes an object of type A:

```
class B
{
        A object;
public:
        //...
        B& operator= (const B& r);
};
```

```
B& B::operator= (const B& r)
{
   if (&r != this)
   {
       object = r.object;
       //...
       return *this;
   }
}
```

When implementing the operator= in the class B, you can call the operator= of the class A!

Automatic type conversion

In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.

In C++, you can achieve this same effect for user-defined types by defining automatic type conversion functions.

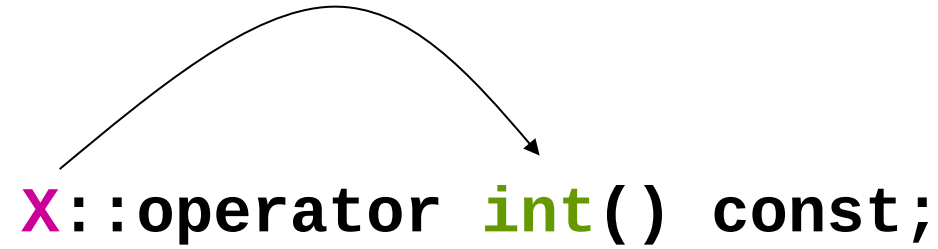
These functions come in two flavours: a particular type of constructor and an overloaded operator.

Operator conversion

You can create a member function that takes the current type and converts it to the desired type using the **operator** keyword followed by the type you want to convert to.

This form of operator overloading is unique because you don't appear to specify a return type – the return type is the name of the operator you're overloading.

For example, to automatically convert an object of type X to an int, the following operator is created:



Operator conversion

```
class X
{   int m_i;

public:
    X(int i) { m_i = i; }
    operator int() const { return m_i; };
};
```

```
X x1(1), x2(2), x3(3);
x3 = x1 + x2;
```

The objects `x1` and `x2` will be converted to integers by the operator: `X::operator int()`.

After adding two integers, the result will be converted back to the type `X` by the constructor.

Constructor conversion

If you define a constructor B that takes as its single argument an object (or reference) of another type A, that constructor allows the compiler to perform an automatic type conversion A->B.

```
class X
{
      int m_i;
public:
       X (int i )   { m_i = i; }
       X (char c)   { m_i = int(c); }
       X (double d) { m_i = int(d); }
       friend const X operator+ (const X& l, const X& r)
      {       return X(l.m_i + r.m_i);          }
};
```

```
X x1(4);
X x2 = x1 + 1.4;
x2 = 'g' + x1;
```

Automatic type conversion

Use automatic type conversion carefully.

Too many type conversions can lead to an ambiguity error:

```
class X
{
      int m_i;
public:
      X(int i ) { m_i = i; }
      operator int () const { return m_i; };

      friend const X operator+ (const X& l, const X& r)
      {
       return X(l.m_i + r.m_i);
      }
};
```

```
X x1(4);
X x2 = x1 + 3;
```

Preventing constructor conversion

There are times when automatic type conversion via the constructor can cause problems. To turn it off, you modify the constructor by prefacing with the keyword **explicit**.

```
X (int i ) { m_i = i; }
```

```
X x5 = 5;
```


```
explicit X(int i ) { m_i = i; }
```

```
X x1(4);
X x2 = x1 + 3;
X x3 = x1 + X(3);
```

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapter 12