LECTURE 7

Overloading Operators Continued

If an operator for a built–in type has a certain number of arguments, you can't overload it with a different number of arguments. For example, you can't make a unary operator binary or a binary operator (like +) ternary. In addition, if you overload the addition operator for some class `Z` (`Z::operator+()`) and the assignment operator (`Z::operator=()`), the compiler will not generate a definition of `Z::operator+=()`. Each operator must be overloaded independently. I want to talk about overloading a few other operators.

Overloading the Prefix and Postfix Operators

Suppose we have a class `USDollar` which has an integer number of dollars and an integer number of cents less than 100. The class declaration might look something like:

```
class USDollar
  {
    friend USDollar operator+(USDollar&, USDollar&);
    friend USDollar& operator++(USDollar&);
    public:
     USDollar(unsigned int d, unsigned int c);
    private:
     unsigned int dollars;
     unsigned int cents;
  };
```

The `operator++()` increments the cents field. If it goes over 100, it increments the dollar field and zeros out the cents. This operator returns a reference to `USDollar` because it changes the value of the provided object.

```
USDollar& operator++(USDollar &s)
  {
   s.cents++;
   if (s.cents >=100)
   {
      s.cents -= 100;
      s.dollars++;
   }
   return s;
  }
```

If you want to overload the prefix operator `++x` separately from the postfix version `x++`, here is the rule. `operator++(ClassName)` refers to the prefix operator and `operator++(ClassName,int)` refers to the postfix operator. The `int` is never used; the argument is simply a dummy used to distinguish between prefix and postfix applications.

The same rule applies to `operator--()`. If you only provide one `operator++()` or `operator--()`, most compilers will use it for both the prefix and postfix versions.

Cast Operator

The cast operator can be overloaded as well. Let's take an example from *C++ for Dummies*.

```
class USDollar
{
public:
  USDollar(double value = 0.0);    //constructor with default value
  //the following function acts as a cast operator
  operator double()
  {
    return dollars + cents / 100.0;
  }
private:
  unsigned int dollars;
  unsigned int cents;
 };
USDollar::USDollar(double value)  //constructor-converts double to USDollar
 {
  dollars = (int)value;
  cents = (int)((value - dollars) * 100 + 0.5);
 }
int main()
 {
  USDollar d1(2.0), d2(1.5), d3;
  //invoke cast operator explicitly...
  d3 = USDollar((double)d1 + (double)d2);
  //...or implicitly
  d3 = d1 + d2;
  return 0;
 }
```

A cast operator is the word `operator` followed by the desired type. The member function `USDollar::operator double()` provides a mechanism for converting an object of class `USDollar` into a `double`. Cast operators have no return type. As the preceding example shows, conversions using the cast operator can be invoked either explicitly or implicitly. Look at the implicit case carefully.

In trying to make sense of the expression `d3 = d1 + d2`, C++ first looked for member function `USDollar::operator+(USDollar)`. When that wasn't found, it looked for the non–member version `operator+(USDollar,USDollar)`. Lacking that as well, it started looking for an `operator+()` that it could use by converting one or the other arguments into a different type. Finally it found a match: by converting both `d1` and `d2` to `doubles`, it could use the intrinsic `operator+(double, double)`. It then has to convert the resulting `double` back to `USDollar` using the constructor.

This demonstrates both the advantage and disadvantage of providing a cast operator. Providing a conversion path from `USDollar` to `double` relieves programmers of the need to provide their own set of operators. `USDollar` can just piggyback on the operators defined for `double`.

On the other hand, it also removes the ability of the programmer to control which operators are defined. By providing a conversion path to `double`, `USDollar` gets all of `double`'s operators whether they make sense or not. In addition, going through the extra conversions may not be the most efficient process in the world. For example, the simple addition just noted involves 3 type conversions with all of the attendant function calls, multiplications, divisions, and so on.

Be careful not to provide 2 conversion paths to the same type. This confuses the compiler.

Overloading Function Call

The function call operator can be overloaded. This typically takes the form `operator()()` or `operator()(arguments)`. These overloaded operators must be member functions. The first pair of parentheses represents the name of an object of the class. The arguments go inside the second pair of parentheses. For example:

```
#include <iostream.h>
#include <math.h>

class Distance{
  private:
   double xc, yc;
  public:
   Distance(double x, double y)          //constructor
    {xc=x;
     yc=y;}
   double operator()()                   //returns a double
     {return sqrt(xc*xc+yc*yc);}

   double operator()(double x)           //returns a double
    {xc=x;                               //reset xc
     return (*this)();}                  //use definition of operator ()()
 };

 main()
   {
    double xx, yy;
    cin >> xx >> yy;
```

```
  Distance d(xx,yy);                          //call constructor
  double oldlength=d();                       //call operator ()()
  xx=5.0;                                     //reset xx
  double newlength=d(xx);                     //call operator ()(double x)
  cout << oldlength << " " << newlength<<endl;
 }
```

Notice that overloading the function call operator has the form `double operator()()` or `double operator()(arguments)` while the cast operator has the form `operator double()`. Remember that the cast operator has no return type.

Type Conversion Using Constructors

C++ can use class constructors to convert an object from one type to another. It will attempt to do this any time it can. However, it can only use constructors with only a single argument (or with all arguments defaulted except for one). This is because of syntax more than anything. Consider the following example:

```
#include <iostream.h>

class Number
  {
   public:
    Number(double x)      //constructor
    {xc=x;}
   private:
    double xc;
  };
void func(Number N)
  {cout << "In function" << endl;}
main()
  {
   double x;
   cin >> x;
   func(x);
  }
```

`func` is defined to take an argument that is a `Number`. However, in `main` `func(x)` is called with an argument that is a `double`. So C++ tries to convert the argument to a `Number`. It notices that the constructor for `Number` essentially converts a `double` into a `Number`. So it uses the constructor to do the type conversion. If ambiguities arise, the compiler will generate an error. For example,

```
#include <iostream.h>

class Number
```

```
  {
   public:
    Number(double x)      //constructor
    {xc=x;}
   private:
    double xc;
  };

class AnotherNumber
  {
   public:
    AnotherNumber(double x)      //constructor
    {xc=x;}
   private:
    double xc;
  };
void func(Number N)
  {cout << "In function 1" << endl;}
void func(AnotherNumber N)
  {cout << "In function 2" << endl;}
main()
  {
   double x;
   cin >> x;
   func(x);
  }
```

To implement `func(x)` C++ could convert `x` to a `Number` and use `func(Number N)`, or to `AnotherNumber` and use `func(AnotherNumber N)`. With no way to resolve the ambiguity, the compiler won't compile the program. To resolve the ambiguity, we add an explicit call to the intended constructor:

```
main()
  {
   double x;
   cin >> x;
   func(Number(x));
  }
```

Notice how the preceding call to the constructor is similar to a cast. Here we have cast a `double` to a `Number`. The similarity is more than superficial. C++ allows this format for specifying a cast. This new format can also be used for intrinsic casts, plus the old format can be used for constructor conversions, as shown in the following:

```
  void fn(int *pI)
  {
    float x = 10.5;
    int i = int(x);                    //same as int i = (int)x;
    func(Number(x));                   //new format for a cast
    func((Number)x);                   //old format for a cast
    double *pD = (double*)pI;          //older format must be used when
                                       //casting pointers
  }
```

Either format is fine. However, you must use the older format when casting from one pointer type to another due to the syntactical confusion that * by itself would cause. Note that a constructor cannot specify an implicit conversion from a user–defined type to a basic type because the basic types are not classes. An overloaded cast operator must be used in this case.

The following rules are very good ideas:

1. Don't make a cast operator and a constructor that do the same jobs. It will confuse the compiler.

2. Don't make more conversion operators than necessary.

If you do not want the constructor to be used implicitly as a conversion operator, then declare the constructor `explicit`. An `explicit` constructor will be invoked only explicitly and implicit conversion will be suppressed. For example,

```
#include <iostream.h>

class Number
  {
   public:
    explicit Number(double x)      //explicit constructor
    {xc=x;}
   private:
    double xc;
  };
void func(Number N)
  {cout << "In function" << endl;}
main()
  {
   double x;
   cin >> x;
   func(x);                //error: no implicit double -> Number conversion
  }
```

Boolean Class

As an example of using constructors to do type conversion, Barton and Nackman introduce the Boolean class on pages 151–153. Boolean constants are true or false. The class is defined as follows:

```
//This is the Boolean.h file.
#ifndef BOOLEANH
#define BOOLEANH

class Boolean{
public:

// Constants
    enum constants{ False = 0, True = 1 };  //writing "false" or "true"
                                            //will clash with built-in types.

// Constructors used for type conversion.
    Boolean()                      {}   // Uninitialized.
    Boolean(int i) :    v(i != 0)  {}   // Initialize v to (i != 0).
    Boolean(float f) :  v(f != 0)  {}   // Initialize v to (f != 0).
    Boolean(double d) : v(d != 0)  {}   // Initialize v to (d != 0).
    Boolean(void* p) :  v(p != 0)  {}   // Initialize v to (p != 0).

// Overloading cast operator to convert Boolean to int.
    operator int() const{ return v; }   // To allow "if (boolean-value)..."

// Overloading negation operator.
    Boolean operator!() const { return !v; }
private:
    char v;
};

#endif
```

We use `char v` in order to store 0(false) and 1(true) in the smallest amount of space in memory. Each of the single argument constructors defines a conversion from a built–in type to a `Boolean`. The `Boolean` object created is initialized to true if the constructor argument is nonzero and to false otherwise. A `void*` is a pointer to an object of unknown type; any pointer can be converted automatically to a `void*` in a manner that converts a null pointer to a null `void*` and a non–null pointer to a non–null `void*`. The cast operator is overloaded to return an `int`. One might worry that "`return v`" might result in a `char`, but the compiler knows to convert the `char` into an `int`.

Here are some examples of how Boolean is used:

```
Boolean b1(Boolean::True);        //calls Boolean(int)
Boolean b2(3);                    //calls Boolean(int)
int* pI=new int(3);               //pI points to an integer initialized to 3
Boolean b3(pI);                   //calls Boolean(void*)
Boolean b4(3.0);                  //calls Boolean(float)
```

Built–In Boolean Type

C++ has a built–in Boolean type called `bool`. This is a relatively new feature of C++, so your book doesn't mention it, but it's described on page 71 of Stroustrup (3rd edition). A `bool` has one of two values: *true* or *false*. A `bool` has the same status as `double` or `int` or `float`; it's a built–in type. So you don't need a Boolean include file if you use `bool`. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
  {
   bool b1=a==b;
   //....
  }
```

If `a` and `b` have the same value, `b1` becomes *true*; otherwise `b1` is *false*.

A function that tests for some condition often returns a `bool`. For example:

```
  bool greater(int a, int b) {return a>b;}
```

By definition, *true* has the value 1 when converted to an integer and *false* has the value 0. Conversely, integers can be implicitly converted to `bool` values: nonzero integers convert to *true* and 0 converts to *false*. For example:

```
  bool b = 8;        //bool(8) is true, so b becomes true
  int i = true;      //int(true) is 1, so i=1
```

In arithmetic and logical expressions, `bool`s are converted to `int`s; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to `bool`, a 0 is converted to *false* and a nonzero value is converted to *true*.

```
   bool a = true;
   bool b = true;
   bool x = a + b;    //a+b is 2, so x=true
   bool y = a|b;      //a|b=1, so y=true
```

A pointer can be implicitly converted to a `bool`. A nonzero pointer converts to a *true*; zero–valued pointers convert to *false*.

The Boolean class defined by Barton and Nackman has the line

```
    enum constants{ false = 0, true = 1 };
```

This won't compile because "`false`" and "`true`" are already defined by the built–in `bool`. So if you change "`false`" to `False` and "`true`" to `True` everywhere in Boolean.h and Boolean.C, it compiles just fine.