Operator Overloading

We have seen examples of operator overloading since we started C++. For example, when we do:

```
cout << "Hello!";
```

Recall that `cout` is an output stream object that is already connected to standard out. << is an abbreviation for calling a function that inserts text into an output stream.

Another example of operator overloading occurs with C++ strings. For example, we can alphabetically compare two strings like this:

```
string s1 = "abc";
string s2 = "def";
if (s1 == s2) {
     cout << "Equal" << endl;
}
```

Here, the == actually calls a function in the string class that compare the strings letter by letter.

Overloading as class members

When we overload operators, we must write a corresponding function for what should happen when we use that operator. There are two ways to write this function – as a member of a class and as a non-member of a class. This section will discuss how to write functions for overloaded operators as a part of a class.

Which operators?

Here is an (incomplete) list of the operators that can be overloaded:

```
+, -, *, /, %
+=, -=, *=, /=, %=
++, --
==, !=, <, >, <=, >=
[]
<<, >>
=
```

When to overload

It's very easy to get carried away with operator overloading. However, overloaded operators that are not intuitive can be very confusing for someone else using your code. For example, if you have a (mathematical) `Vector` class, it makes sense to overload the + operator so that you can

add the components of two vectors. However, it makes no sense to overload the % operator – that doesn't correspond to any natural vector operation.

Prototypes
The prototype for an overloaded member function looks like this:

```
//Do not put a space between "operator" and the sign
returnType operator sign(params);
```

Here, `sign` is the operator you're overloading, like +. Consider the following `Vector3` class:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(int, int, int);
          Vector3(void);
};

Vector3::Vector3(int x, int  y, int z) {
     this->x = x;
     this->y = y;
     this->z = z;
}

Vector3::Vector3(void) {
     x = 0;
     y = 0;
     z = 0;
}
```

Now, suppose we want to overload the + operator to be able to add another vector to THIS vector and return the result. Here's what the prototype would look like (it would go inside the class definition):

```
Vector3 operator+(const Vector3&);
```

Notice that we are passing the `Vector3` object by *constant reference* to save space.

Implementation
Here's what the implementation would look like:

```
Vector3 Vector3::operator+(const Vector3& v) {
     Vector3 result(x+v.x, y+v.y, z+v.z);
     return result;
```

```
}
```

Note that this is exactly the same format that we've used before when implementing functions. The only difference is that the name of this function is `operator+`, since we're overloading the + operator.

Using overloaded operators

We can use an overloaded operator just like a normal function. For example, if we have two `Vector3` objects:

```
Vector3 v1(1, 2, 3);
Vector3 v2(4, 5, 6);
```

Then we could add them together by calling the `operator+` function:

```
Vector3 result = v1.operator+(v2);
```

Now, result would have the values (5, 7, 9). However, we could also compute the result this way:

```
Vector3 result = v1 + v2;
```

The compiler converts this to `v1.operator+(v2)`.

More examples

In this section, we will overload more operators for the Vector3 class. Specifically:

- == (to compare if two vectors have the same x, y, and z)
- * (to compute the dot product between vectors)
- [] (to access x, y, and z as if the vector was an array)

Here is the updated class definition:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(int, int, int);
          Vector3(void);
          Vector3 operator+(const Vector3&);
          bool operator==(const Vector3&);
          int operator*(const Vector3&);
          int& operator[](int index);
};
```

And here is the updated class implementation:

```
Vector3::Vector3(int x, int  y, int z) {
     this->x = x;
     this->y = y;
     this->z = z;
}

Vector3::Vector3(void) {
     x = 0;
     y = 0;
     z = 0;
}

Vector3 Vector3::operator+(const Vector3& v) {
     Vector3 result(x+v.x, y+v.y, z+v.z);
     return result;
}

bool Vector3::operator==(const Vector3& v) {
     if (v.x==x && v.y==y && v.z==z) return true;
     else return false;
}

int Vector3::operator*(const Vector3& v) {
     return x*v.x + y*v.y + z*v.z;
}

int& Vector3::operator[](int index) {
     if (index == 0) return x;
     if (index == 1) return y;
     if (index == 2) return z;

     throw "Invalid array index";
}
```

Here's an example of using == and *:

```
Vector3 v1(1, 2, 3);
Vector3 v2(4, 5, 6);
int val;

if (v1 == v2) cout << "Equal" << endl;

val = v1 * v2;
```

The overloaded array operator is a bit trickier – notice that it returns a reference either `x`, `y`, or `z`. This is because we want to be able to do something like this:

```
v1[0] = 4;
```

To change the x-value in `v1` to 4. This is equivalent to:

```
v1.operator[](0) = 4;
```

So, we need to be able to change the return value of a function. If we return a reference to either `x`, `y`, or `z`, then the function call becomes an alias for `x`, `y`, or `z`. We can then update this alias (the function call) to have a new value, which really changes either `x`, `y`, or `z`.

Assignment operator

The assignment operator (=) is automatically overloaded for all classes. This function takes another object of this class type, and initializes the instance variables of the current object to those of the parameter object.

This allows us to do something like this:

```
Vector3 v(1, 2, 3);
Vector3 newVect = v;
```

Here, the second line creates an entirely new `Vector3` object and sets its `x`, `y`, and `z` instance variables to the `x`, `y`, and `z` variables in `v`.

Even though the assignment operator is automatically overloaded, it is often useful to implement it yourself. For example, if one of the instance variables was an array, the default overloaded function would set the new array instance variable equal to the original one. It would not create a new array and copy the values individually, which is want you would probably want.

If you have a class that uses dynamic memory, then you probably want to create your own overloaded assignment operator. This operator should return a reference to the updated object, and should take the "copy" object as an object. For the `Vector3` class, here's what the prototype would look like:

```
Vector3& operator=(const Vector3&);
```

And this would be the implementation:

```
Vector3& operator=(const Vector3& v) {
     this->x = v.x;
     this->y = v.y;
     this->z = v.z;
```

```
        //return the updated object
        return *this;
}
```

Overloading as non-members

Overloaded operators do not have to be implemented as members (inside) of a class – and sometimes, it's better if they aren't. If you decide to overload an operator as a function that is separate from a class, then you should always pass an additional argument to the overloaded function.

For example, the operator+ in the Vector3 class took one Vector3 object because it added THIS object and the parameter object. If an overloaded function is outside a class, there is no "this" object. In this case, we would pass operator+ two Vector3 objects.

Prototypes

The prototype for an overloaded operator function that is not a member of a class should be declared in the same .h file as the class it's operating on (right below the class definition). For example, here is the entire `vector3.h` header file, complete with a non-member prototype for adding two vectors:

```
//vector3.h

class Vector3 {
      public:
            int x, y, z;
            Vector3(int, int, int);
            Vector3(void);
};

//prototype for non-member operator+
Vector3 operator+(const Vector3&, const Vector3&);
```

Implementation

The implementation of a non-member overloaded operator goes in the .cpp file of the class implementation. Just remember that this function is not part of the class, so we will not include the `Vector3::` scope operator. Here's what the vector3.cpp file will look like:

```
//vector3.cpp

Vector3::Vector3(int x, int y, int z) {
      this->x = x;
      this->y = y;
      this->z = z;
```

```
}

Vector3::Vector3(void) {
     x = 0;
     y = 0;
     z = 0;
}

//operator+ implementation – NOT part of the class
Vector3 operator+(const Vector3& v1, const Vector3& v2) {
     Vector3 result(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
     return result;
}
```

Notice that we can access `x`, `y`, and `z` from `v1` and `v2` because they are now PUBLIC instance variables. If they were private variables, then we would be unable to access them since `operator+` is not part of the `Vector3` class. In the next section, we will learn how to create *friend* functions. This will allow us to access private variables even if we're not part of a class.

Using non-member functions

We use non-member overloaded operators in the same way we did operators that were part of the class, but the compiler converts them differently.

For example, I can still add vectors like this:

```
Vector3 v1(1, 2, 3);
Vector3 v2(4, 5, 6);

Vector3 result = v1 + v2;
```

However, now the compiler converts the above call to `operator+` like this:

```
Vector3 result = operator+(v1, v2);
```

Example: Money class

In this example, we will write a class that represents money in dollars and cents. Here is the class declaration:

```
class Money {
     public:
          int dollars, cents;
          Money(int, int);
          Money(int);
```

```
            Money(double);
            Money(void);
            double getAmount(void);
};

Money operator+(const Money&, const Money&);
Money operator-(const Money&, const Money&);
```

Our money class holds the dollars and cents for a particular money amount. It has two overloaded non-member functions that can add and subtract amounts of money.

Here is the implementation:

```
Money::Money(int d, int c) {
     dollars = d;
     cents = c;
}

Money::Money(int d) {
     dollars = d;
     cents = c;
}

Money::Money(double amount) {
     if (amount >= 0) {
          dollars = (int)(amount+1) - 1;
     }
     else {
          dollars = (int)(amount-1) +1;
     }

     cents = amount-dollars;
}

Money::Money(void) {
     dollars = 0;
     cents = 0;
}

double Money::getAmount(void) {
     return dollars + cents/100.0;
}

Money operator+(const Money& m1, const Money& m2) {
     Money result(m1.getAmount() + m2.getAmount());
     return result;
}
```

```
Money operator-(const Money& m1, const Money& m2) {
     Money result(m1.getAmount() - m2.getAmount());
     return result;
}
```

Now, we can use the money class like this:

```
Money m1(2, 75);
Money m2(5.14);

Money plusMoney = m1 + m2;
Money minusMoney = m2 - m1;
```

Automatic type conversion

It would be nice to be able to use the money class like this as well:

```
Money m(5.42);

Money plusFive = m + 5.00;
```

For this to work, the compiler must have some way of converting 5.00 into a `Money` object, since the `operator+` function takes two `Money` objects.

In fact, the compiler can do just this using something called *automatic type conversion*. The compiler sees the call to `operator+`, and sees that the function takes two `Money` objects. It sees that `m` is a `Money` object, but that 5.00 is not. So, it sees if there is a `Money` constructor that takes a double as an argument – there is, so the compiler converts 5.00 into a `Money` object.

In short, you can pass something of type A to a function that takes an object of type B if class B has a constructor that takes a single parameter of type A. This means that there is a way to convert A into B.

Suppose for a moment that the `operator+` function was overloaded as a member function instead. Then the above call to `operator+` would translate to this:

```
//Adding 5.00 if operator+ is a member function
Money plusFive = m.operator+(5.00);
```

This would still work fine – the compiler would convert 5.00 into a `Money` object and complete the function call. However, suppose we tried to do this instead:

```
Money plusFive = 5.00 + m;
```

We would expect the same results – it shouldn't matter in which order we add objects – but with a member `operator+` function, the call would look like this:

```
//Adding 5.00 first if operator+ is a member function
Money plusFive = 5.00.operator+(m);
```

This makes no sense. The compiler will not be able to figure out that 5.00 is supposed to be a `Money` object. However, with `operator+` as a non-member function, it looks like this:

```
//Adding 5.00 first if operator+ is a non-member function
Money plusFive = operator+(5.00, m);
```

which is still fine – the compiler converts the 5.00 argument into a `Money` object.

In summary, if you want to take advantage of automatic type conversion, you should overload functions as non-members.

Member v. non-member

We have now seen how to overload operators as both class member functions and non-member functions. Both have their advantages.

When to overload as a class member:

- Array indexing overload, []
- Copy operator, =
- Most of the time when you are operating on a single object

When to overload as a non-member:

- Most of the time when you are operating on two object
- When you want to take advantage of automatic type conversion

Overloading streams

The last kind of operator overloading we will discuss is how to overload input and output operators, << and >>. For example, if we have a string variable then we can print it like this:

```
string s = "Hello";
cout << s;
```

This is possible because the C++ string class overloads the << operator, which provides instructions for adding each character to the output stream. It is a good idea to overload << in all your data classes so that you can print their contents for debugging.

To demonstrate how this works, let's build an `Integer` class. This class will contain an int integer variable, and have constructors that initialize this variable. (This is a silly class, but it will more clearly demonstrate the concept of overloading streams.)

Here is the `Integer` class definition:

```
class Integer {
     public:
          int x;
          Integer(void);
          Integer(int);
};

ostream& operator<<(ostream&, const Integer&);
istream& operator>>(istream&, Integer&);
```

A couple of things to note:

- << and >> MUST be overloaded as non-member functions
- << and >> take the current output stream or input stream, update it, and return the resulting stream.
- The notation looks scary, but recall that `cout` is an `ostream` and `cin` is an `istream`

Here is the implementation:

```
Integer::Integer(void) {
     x = 0;
}

Integer::Integer(int val) {
     x = val;
}

ostream& operator<<(ostream& out, const Integer& intObj) {
     out << intObj.x;
     return out;
}

istream& operator>>(istream& in, Integer& intObj) {
     in >> intObj.x;
     return in;
}
```

Again, we use the `ostream` and `istream` parameters just like we use `cout` and `cin`. Then, we return the updated stream.

We can now use `Integer` objects like this:

```
Integer int1(4);
Integer int2;
```

```
cout << "Enter an integer: ";

//Reads the integer into int2.x
cin >> int2;

//Adds int1 and int2 to the standard output stream
cout << int1;
cout << int2;
```