Operator Overloading

Operators are designed to clean up the syntax of certain functions. They allow us to work with C++ types in the way we are used to manipulating them in mathematics. For example, in order to add two numbers and assign that result to a third value, we would prefer to write `a = b + c` rather than `a.set(b.add(c))`.

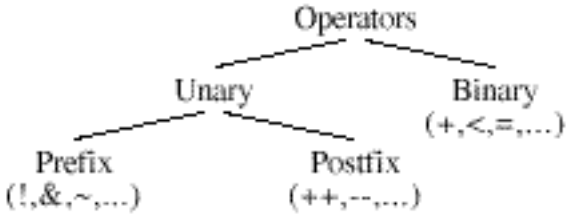
Using operators prevents our code from looking too much like LISP with parentheses flying around everywhere. C does a good job at supplying operators for built-in types such as integers and floating-point numbers. However, if we had a type that represented a complex number, a fraction, or even a character string in C, we would have to use function-style syntax for manipulation. C++ allows for the definition of operators operating on user-defined types. This feature is called **operator overloading**, because a single operator can handle multiple argument types.

Readings from Eckel:
Chapter 12: Operator Overloading

Readings from Deitel[2]:
Chapter 8: Operator Overloading

Types of Operators

There are several types of operators in C and C++. They can be categorized as **unary** or **binary**, depending if they accept one or two arguments. Unary operators can be **prefix** or **postfix**, depending on if the operator appears before or after the object in an expression. The taxonomy of operators looks something like this:



Some operators can be either unary or binary. For instance, the `*` operator can be used as unary operator for dereferencing, or as a binary multiplication operator. Some unary operators can be either prefix or postfix, such as `++`, or `--`. The `[]` operator, which is used for indexing, is also a binary operator, even though one operand appears within the operator. `new` and `delete` are also unary prefix operators which can be overloaded. (The array forms of new and delete are different from the non-array forms,

so there are a total of four of these operators.) For a binary operator, we call the expression on the left side of the operator the left-hand operand and the expression on the right side of the operator the right-hand operand.

Syntax

Suppose we had a class which represented large numbers. This `BigNum` class might store a number as an array of digits so that it could represent any number, no matter how big it became. A definition of this class might look something like this:

```
class BigNum
{
   public:
      BigNum(int value = 0);
      BigNum(const string& numString);
      ~BigNum(void);
      ...
   private:
      vector<char> digits;
};
```

We want to manipulate `BigNum` objects by comparing them, adding them, etc. Instead of having member functions such as `isEqual`, or `addTo`, it would be nicer to be able to simply compare them and perform arithmetic using the appropriate operators. An operator can be defined within a class as a member function, or it can be defined as a global function which is not part of any class. The number of function arguments for these is shown in the table below

Operator "arity"	Member Function	Global Function
unary	0	1
binary	1	2

The reason that member functions and global functions have different numbers of parameters is because there is an "extra" parameter for the member function, which is the class instance on which the operator is being invoked. For a binary operator, this is always the left-hand operand. For example, operators could be defined within the class like this:

```
class BigNum {
   ……
   ……
   const BigNum operator+(const BigNum& num);
   bool operator==(const BigNum& num);
   ……
};
```

The first addition operator adds two `BigNum's`, the second operator compares to `BigNums` for equality. You can also declare operators outside of the class like this:

```
const BigNum operator+(const BigNum num1, const BigNum& num2);
bool operator==(const BigNum num1, const BigNum& num2);
```

An operator which is a member function is implemented just like other member functions, and an operator which is a global function is implemented like other global functions. While you can provide either a member function or a global function for a particular operator, you cannot provide both for the same argument types.

Mixing Types

Suppose you want to be able to declare a `BigNum` object and mix the `BigNum's` freely with integers. For instance, you'd want to be able to write:

```
BigNum height, width;
.. .. ..
if (width == 3) // OK with global or member function, or with conversion
if (3 == width) // Only possible with a global function or with conversion
```

There are three ways to provide the first comparison:

- Provide an == operator as a member function accepting an integer.
- Provide an == operator as a global function accepting a `BigNum` and an integer.
- Provide an == operator as a global function accepting a `BigNum` and a `BigNum` and provide a constructor that converts an integer into a `Bignum`.[1]

There are only two ways to provide the second comparison. This is because an operator which is a member function must be a member of the left-hand operand's class. In this case, the left-hand operand is an integer, and there is no class to which we can add the operator function. This leaves us with two options:

- Provide an == operator as a global function accepting an integer and a `BigNum`.
- Provide an == operator as a global function accepting a `BigNum` and a `BigNum`, and provide a constructor converting an integer into a `BigNum`.

There are trade-offs to these methods. Using member functions can be easier because member functions can access `private` data members directly. Using global functions without conversions can be faster, but requires duplicating code which can be error-prone. Using a **most-general** global function with conversions can be slower, but avoids duplication of code.

Comparison

Comparison operators generally have a return type of `bool`. Comparison operators include `!`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, and `||`. When using a binary comparison operator and

[1] The compiler will silently convert the integer into a `BigNum` by calling the constructor...this all happens automatically!

comparing to built-in types, they should be defined as global functions so that you can compare without worrying about the order of arguments. For the `BigNum` class, we might define the following equality operators:

```
bool operator==(const BigNum& num1, const BigNum& num2);
bool operator==(const BigNum& num1, int num2);
bool operator==(int num1, const BigNum& num2);
```

Or, we might simply define the first equality operator and allow the compiler to call the `BigNum` constructor to convert an integer to a `BigNum` before the comparison is done. Note that the `BigNum` arguments can be defined to be references in order to avoid copying.

Arithmetic

There are numerous arithmetic and logical operators, such as `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `|`, `&`, `^`, and `~`. They generally return an object of the type of the class, and **not** a reference.

We might define an addition operator for the `BigNum` class like this:

```
const BigNum operator+(const BigNum& num1, const BigNum& num2);
```

Note that we return a `const BigNum` instead of just a `BigNum`. This is because we don't want the result to be changed — it is usually a hidden temporary. For example, if the `BigNum` class could read itself in from a stream (we'll learned how to do this soon), we would not want to allow the following code:

```
BigNum height, width;
cin >> (height + width); // Should be illegal
```

We could implement the addition operator like this:

```
const BigNum operator+(const BigNum& num1, const BigNum& num2)
{
   BigNum result;
   // Do the addition...
   return result;
}
```

Unary arithmetic operators have two objects involved — the operand and the result. Binary arithmetic operators have three objects involved — the left-hand operand, the right-hand operand, and the result. We don't want to return a reference as the return type, because that would be returning a reference to a local variable of a function — a very bad idea.

Arithmetic + Assignment

There are operators which combine arithmetic and assignment, such as +=, -=, *=, /=, %=, <<=, >>=, |=, &=, and ^=. These involve two objects — the right-hand operand, and the left-hand operand which is also the return value. The arithmetic-assignment operators generally return a reference to the class type to avoid copying, and are normally implemented as member functions because the left-hand operand must always be the type of the class. The += operator would look something like this for the `BigNum` class:

```
BigNum& BigNum::operator+=(const BigNum& num)
{
   // Do the addition...
   return *this;
}
```

Increment and Decrement

The ++ and -- operators present an interesting challenge because they can be either prefix or postfix operators. That is, you could do the following:

```
BigNum number;
number++;
++number;
```

Because you'll probably want the prefix operator to return something different than the postfix operator, C++ handles this wrinkle in a somewhat non-intuitive way:

- The ++ operator defined as a member function with no parameters is prefix.
- The ++ operator defined as a global function with one parameter is prefix.
- The ++ operator defined as a member function with an integer parameter is postfix.
- The ++ operator defined as a global function with two parameters is postfix. The second parameter must be an integer.

Basically, any time the compiler calls the prefix operator, it uses only the left-hand and right-hand operands. However, when the compiler calls the postfix operator it adds a hidden integer parameter to distinguish the call from the prefix operator. Generally, the prefix operator should return the value of the object **before** the operation is applied. The postfix operator should return the value of the object after the operation is applied.

Subscript

The [] operator is generally used for indexing. The [] operator must be defined as a member function. Usually the index provided it is an integer, but C++ allows any type to be provided for the index. The [] operator is a common place where const / non-const overloading occurs. Suppose we wanted to define a class which wraps around an array of integers, if for no other reason than to provide bound checking on access:

```
class IntArray {

   public:
      IntArray(int size);
      IntArray(const IntArray& array);
      const IntArray& operator=(const IntArray& src);
      ~IntArray();

   private:
      int *array;
      int size;
};

IntArray::IntArray(int size)
{
   this->size = size;
   this->elems = new int[this->size];
   for (int i = 0; i < this->size; i++)
      this->elems[i] = 0;  // do deep initialization
}

IntArray::IntArray(const IntArray& array)
{
   this->size = array.size;
   this->elems = new int[this->size];
   for (int i = 0; i < this->size; i++)
      this->elems[i] = array.elems[i];
}

IntArray::~IntArray()
{
   delete[] elems;
}

const IntArray& IntArray::operator=(const IntArray& src)
{
   if (this != &src) {
      delete this->elems;
      this->size = src.size;
      this->elems = new int[this->size];
      for (int i = 0; i < this->size; i++)
         this->elems[i] = src.elems[i];
   }

   return *this;
}
```

We would like to provide a way to access the elements directly so that our array class responds to the same syntax that an ordinary array would. We would want a constant

IntArray to allow read-only access to an element, but a non-constant IntArray would allow read / write access to an element. We can do this by returning an integer in the read only case and an integer reference in the read / write case:

```
class IntArray {

   public:
      IntArray(int size);
      IntArray(const IntArray& array);
      ~IntArray();

      const IntArray& operator=(const IntArray& src);
      int operator[](index i) const;
      int& operator[](index i);

   private:
      int *array;
      int size;
};
```

Any time the [] operator is called on an IntArray object which is non-const will return a reference to one component of the vector. Modifying that reference will modify the IntArray itself. Using the [] operator on a const IntArray simply returns a copy of the array component.

```
#include <assert.h>
int IntArray::operator[](int i) const
{
   assert(i >= 0 && i < size);
   return elems[i];
}

int& IntArray::operator[](int i)
{
   assert(i >= 0 && i < size);
   return elems[i];
}
```

Stream I/O

We often want to add the ability to read a user-defined type from a stream or output a user-defined type to a stream. We can do this by defining an operator which takes an istream or ostream as the left-hand operand and our type as the right-hand operand. Because we don't want to change the standard library code in order to add this operator as a member function, we do this by adding a global function for the operator. For instance, to read a BigNum from a stream, we could define this operator:

```
istream& operator>>(istream& input, BigNum& num);
```

To output a BigNum to a stream, we could define this operator:

```
ostream& operator<<(ostream& output, const BigNum& num);
```

We would implement this operator similar to this:

```
ostream& operator<<(ostream& output, const BigNum& num)
{
   // Output each digit...
   return output;
}
```

The reason the `istream`/`ostream` is generally returned is so that you can chain inputs and outputs. When you write:

```
cout << "It is " << 32 << "degrees out!";
```

It is like writing:

```
(((cout << "It is ") << 32) << "degrees out!");
```

Where each operator is returning the stream after it adds its output which is used as the left-hand operand of the next output operator.

Friends

When implementing operators which are global functions, you normally have zero access to `private` data. While you may want to add accessors for this data, that doesn't always make sense — you don't want everyone to access the data, but you do want to be able to access it in your operator function. One way to solve this problem is by the use of friends. A `friend` is a way of specifying that a single function or an entire class can have access to the `private` function and data members. If we wanted the output operator for our `BigNum` to access the digits directly, we would specify the following friend function in the class definition:

```
class BigNum
{
   ...
   friend ostream& operator<<(ostream& output, const BigNum& num);
   ...
};
```

We still have to declare the output operator function and implement it elsewhere. We can make either a global or a member function a friend. To make a member function a friend, we use the class specifier along with the member function name. If we wanted to provide an iterator for the IntVector class and wanted to allow that iterator class to have full access to the IntVector class, we would do the following:

```
class IntArrayIterator
{
   ...
};
```

```
class IntArray
{
   ...
   friend IntArrayIterator;
   ...
};
```

Style

There are some important style considerations when using operator overloading. The most important one is to respect the intended meanings of the operators—don't be overloading the addition operator to do division. Also, even though you have the flexibility to return whatever type you want to from an overloaded function, you should always map the return types to match whatever is returned by the built-in operators you are overloading. Don't return an `int` from `IntArray::operator==`; instead, return the `bool` one would expect to get back.

Limitations

Operator overloading has certain restrictions to keep programmers from doing extremely confusing things and to make writing a C++ compiler at least somewhat bearable.

- All operator overloading functions must contain at least one user-defined type. This is so you cannot redefine the basic meaning of operators if they don't apply to a user-defined type. C++ is designed to be an extensible, but not mutable language.
- You cannot overload `.`, `*.`, `::`. Trust me when I say you'll never feel confined by this.
- You cannot define any additional operators. Otherwise, even the best compiler would have a though time checking for errors while parsing your program, since everything it sees could potentially be some nifty operator you define in some other file.
- You cannot change the precedence or associativity of an operator. This also simplifies the parsing. Remembering the order of operations is hard enough; letting the programmer tinker with it would make it any easier.