Review

- Describe pass-by-value and pass-by-reference
- Why do we use pass-by-reference?
- What does the term **calling object** refer to?
- What is a **const member function**?
- What is a **const object**?
  - How do we initialize it?
- What is **const member data**?
  - How do we initialize it?
- By what mechanism is **const** enforced?

Operator Overloading

Rethinking Operators

There are many operators available that work on built-in types, like int or double (+, -, *, /, ==, !=, <<, >>).

We could imagine these operators being implemented by function calls made automatically by c++,

For example:

```
int x = 1 + 2;
```

Could automatically be replaced by a function call that looks like:

```
int x = operator+(1, 2)
```

With the following prototype:

```
int operator+(int p1, int p2); /* returns sum of p1 and p2 */
```

Rethinking Operators, cont.

Another example:

```
double var = 1.1;
double x = 4.3 * 2.1 - var ;
```

Could automatically be replaced by function calls that look like:

```
double x = operator-(operator*(4.3, 2.1), var)
```

With the following prototypes:

```
double operator*(double p1, double p2); /* returns product of p1 and p2 */
double operator-(double p1, double p2); /* returns difference of p1 and p2 */
```

- C++ provides **operator overloading** to allow the familiar operators to be used with user-defined types (such as objects).

Motivation

Example 1:

Consider the following use of arithmetic operators:

```
int x = 3, y = 6, z;
float a = 3.4, b = 2.1, c;
z = x + y;
c = a / b;
```

Now consider if we wanted to add fraction objects in this familiar way (this is pseudo-code):

```
1/2 + 1/3;    /* evaluates to 5/6 */
2/3 - 1/3;    /* evaluates to 1/3 */
```

We could achieve this using the Fraction class as such:

```
Fraction n1, n2, n3;
n3 = n1 + n2;          /* will the compiler accept this? */
```

- It should be clear that this would not make sense to the compiler by default. Fraction is a programmer-defined type. How would the computer know about common denominators, etc?
- These code statements would be nice to use, however, because it's the same way we use other numeric types (like int and double).

Motivation, cont.

Example 2:

Consider screen output, we know the following is legal:

```
int x = 5, y = 0;
cout << x << y;
```

But what about this?

```
Fraction f(3,4);
cout << "The fraction f = " << f << '\n';
```

- Again, it is clear the compiler has no idea how a Fraction object should be output to the screen.

We can use operator overloading to provide a function to implement the desired behavior of such an operation.

Operator Overloading Format

An operator overload is just a function. This means that it must be created with a return type, a name, and a parameter list

The name of an operator is always a conjunction of the keyword ***operator*** and the operator symbol itself.

operator+, operator++, operator<<, operator==, etc.

So the format of an operator overload declaration is just like that of a function, with the keyword operator as part of the name:

returnType *operator*OperatorSymbol (parameterList);

Example:

The string class has overloaded operator<< to allow output of strings using cout. The compiler will call this function automatically when we use the << operator or we can call it explicitly.

See example1.cpp.

Operator Overloading Details

Operator overloading has the following limitations:

- Overloading an operator cannot change its precedence.

  a+b*c will always be equivalent to a+(b*c)

- Overloading an operator cannot change its associativity.

  a+b+c will always be equivalent to (a+b)+c

- Overloading an operator cannot change its "arity" (number of operands).

  The function that performs a+b will always have 2 parameters, we cannot do a+b+c in one function

- It is not possible to create new operators, only new versions of existing ones.
- Operator meaning on the built-in types cannot be changed.

Operator Overloading Details, cont.

Some operators can be written as member functions of a class.

Some operators can be written as stand-alone functions.

- It's common to use the *friend* keyword on these.

Some operators can be written either way (your choice).

A **binary operator** has two operands (eg. f1 + f2).

- Written as a stand-alone function, both operands would be sent as parameters, so it would be a function with two parameters
- Written as a member function, the first operand would be the calling object, and the other would be sent as a parameter (i.e. a function with one parameter)

- A **unary operator** has one operand (eg. f++).
  - As a stand-alone function, the operand is sent as a parameter
  - As a member function, one calling object, no parameters

Overloading Arithmetic Operators

Lets see how we could add fractions using the '+' operator by creating a member function.

What we are shooting for:

```
Fraction f1, f2(1, 2), f3(3, 4);
f1 = f2 + f3;
```

The second line translates to:

```
f1 = f2.operator+(f3);  /* member overload */
        OR
f1 = operator+(f2, f3); /* stand-alone overload */
```

Overloading Arithmetic Operators, cont.

Lets start by implementing the + overload as a member function, we must add the following function to the Fraction class declaration:

```
Fraction operator+(const Fraction &rf) const;
```

The member function definition:

```
Fraction Fraction::operator+(const Fraction &rf) const {
    Fraction res;
    res.numerator = (numerator * rf.denominator)
                    + (rf.numerator * denominator);
    res.denominator = (denominator * rf.denominator);
    return res;
    }
```

- That's it! See member_overload example.

Overloading Arithmetic Operators, cont.

Now lets see that same overload implemented as a stand-alone function. We must make the stand-alone function a friend of the Fraction class to allow access to private members:

```
friend Fraction operator+(const Fraction &lf, const Fraction &rf);
```

The stand-alone function definition:

```
Fraction operator+(const Fraction &lf, const Fraction &rf) {
    Fraction res;
    res.numerator = (lf.numerator * rf.denominator)
                + (rf.numerator * lf.denominator);
    res.denominator = (lf.denominator * rf.denominator);
    return res;
    }
```

- Pretty simple. See alone_overload example.

Overloading Comparison Operators

The comparison operators can also be overloaded either as stand-alone functions or member functions

Consider the Equals function example:

```
friend bool Equals(const Fraction& f1, const Fraction& f2);
```

We can easily write this as an operator overload:

```
friend bool operator== (const Fraction& f1, const Fraction& f2);
```

Here are corresponding sample calls:

```
Fraction n1, n2;
if (Equals(n1, n2))
    cout << "n1 and n2 are equal";
```

Contrast with this:

```
Fraction n1, n2;
if (n1 == n2)
    cout << "n1 and n2 are equal";
```

Overloading Comparison Operators

To get a full set of comparison operations, you could overload all 6 of them:

operator ==

operator !=

operator <

operator >

operator <=

operator >=

Overloading Stream Operators

The stream insertion (<<) and extraction (>>) operators will not automatically work with your class (should be obvious by now).

Since we know the stream insertion operator is overloaded for the string class, here is what the overload prototype is:

```
ostream& operator<<(ostream &os, const string& s);
```

This looks a little wierd, lets ask the following:

- Why is the first parameter an ostream&?
- Why is the first parameter not const?
- Why does the function return an ostream&, shouldn't it be void?
- This is a stand-alone function, could this be a member function?

Overloading Stream Operators, cont.

Q: Why is the first parameter an ostream&?

cout is an object of type ostream. Since the call is of the form cout<<str, the first parameter to the overload is cout (the second is str).

Q: Why is the first parameter not const?

The cout object can be thought of as representing the scsreen. We are changing the screen in the function (with output) ergo the cout object will be changed.

Q: Why does the function return an ostream&, shouldn't it be void?

This is to support cascading calls like this:

cout<<s1<<s2<<s3; => operator<<(cout, s1) <<s2<<s3;

=> cout<<s2<<s3; => operator<<(cout, s2) <<s3;

=> cout<<s3; => operator<<(cout, s3);

=> cout; => ;

Overloading Stream Operators, cont.

Q: This is a stand-alone function, could this be a member function?

No, because the left parameter is of type ostream, the calling object would have to be an ostream (hence it would have to be a member function of the ostream class). Since the string (and Fraction) class are created independently of ostream, it is infeasible that the author of the ostream class would have implemented it as a member function.

Overloading Stream Operators, cont.

Here are the function prototypes for overloading the insertion and extraction operators for the Fraction class:

```
ostream& operator<<(ostream &os, const Fraction &f);
istream& operator>>(istream &is, const Fraction &f);
```

And possible definitions (assuming they are friends of Fraction):

```
ostream& operator << (ostream& os, const Fraction& f)  {
  os << f.numerator << '/' << f.denominator;
  return os;
}
istream& operator >> (istream& is, const Fraction& f)  {
  char slash;
  is >> f.numerator >> slash >> f.denominator;
  return is;
}
```

- See stream_fraction example.