Operator Overloading … and more: **this** Pointer, **friend** Functions

[Much of the material in these notes is derived from the Hubbard book]

Operator overloading is exactly like function overloading. In fact, the expression A+B is actually a call to a function that adds the two operands. The function invoked is called `operator+` and takes as arguments the two operands, A and B. The operator+ function on integer operands A and B would work differently from the same function on two double operands, so the operator+ function is already overloaded.

What if we wish to be able to add two arrays Aarray and Barray? We can't do it because the operation is not defined for the array structure – Aarray and Barray are actually pointers. But we can overload the + operator if we wish to include the array addition operation.

Same for: ==, <<, etc.

The C++ operators (for example, + or <=) are defined automatically for the fundamental types (e.g. + works on int, float, etc.). When you create a class you sometimes need to define operations for the class, and sometimes you would like to use the "standard" operators rather than, say, define an add(x, c) function. The standard operators are not automatically defined for these new classes (i.e., programmer-defined types) and objects creased on these classes cannot make use of these operators in (say) expressions. These operators may work slightly differently on the structured objects of the class than they do on data objects of the fundamental types.

The nice thing about C++ (as opposed to, say, Pascal, COBOL, etc.) is that we can redefine almost any operator to work on the new type that we create by defining a class.

```
//fraction class definition – (source: Friedman/Koffman 2007)
class fraction{
public:
fraction();
fraction(int);
fraction (int, int);
//multiply fractions
fraction multiply (fraction f1);
void readfrac();
void displayfrac();
private:
//data members (attributes)
int num;
int den;
}; //end fraction class definition

//main driver
int main(){
fraction f1, f2;
fraction f3;  //output operand – result of arithmetic operation
cout<<"enter first fraction";
f1.readfrac();
cout<< "enter second fraction";
f2.readfrac();
f3 = f1.multiply(f2); //notice we are using the assignment operator
f1.displayfrac(); cout << " * ";
f2.displayfrac(); cout << " = ";
f3.displayfrac();
}//end main

//fraction class implementation
//3 constructors
fraction::fraction(){
num=0; den=1;
}
fraction::fraction(int n){
num=n; den=1;
}
fraction::fraction(int n, int d){
num=n; den=d;
}
//multiply 2 fractions
fraction fraction::multiply(fraction); {
fraction temp(num*f.num, denom*f.denom);
return temp;
}
//read a fraction
void fraction::readfrac(){
char slash;
cout<<"enter numerator / denominator: ";
cin>> num >> slash >> den;
}
//display a fraction
void fraction::displayfrac(){
cout<<num<<'/'<<den;
}
```

Two issues with the above:

How can we overload the multiply operator – `operator* -` instead of creating the new method?

How can we just multiply two fractions instead of telling one fraction (f1) to multiply itself by another? (use a trusted `friend`)

From here using Rational class instead of Fraction class. Same thing – notes still under construction

Assignment Operator

The assignment operator is one that *is* automatically overloaded when we create a new class. It is used to copy one object to another.

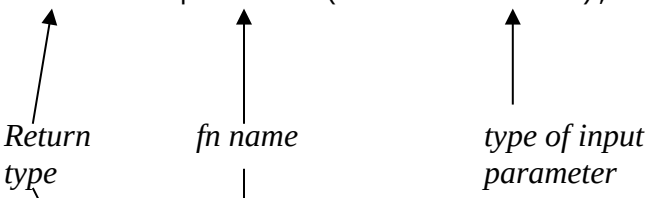
The assignment operator is used to copy the value of one object to another. Just like the default constructor and copy constructor it is automatically overloaded for our new class even if we do not explicitly define it; However, just like the constructors we can explicitly define it if we wish. Ex.

```
class Rational{
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   Rational& operator=(const Rational&);
private:
   int num;
   int den;
};
```

The function name for the assignment operator is "operator=". So *operator overloading* is actually no different from *function overloading*.

The prototype:

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



and in general, where T is a placeholder that stands for "any type," the syntax is:

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

The & is for efficiency: the function does not make a local copy of the object. The const is for safety: the function cannot alter the original.

If x and y are both declared as type Rational, then the expression x=y will assign the values of the data members of y to the corresponding data members of x. Then, why do we need the return type (why not void?).

If x, y, and z are all declared as type Rational, we can chain assignment operations as: x = y = z;

We can only do that if the first expression evaluated, y = z, returns a value that can then be used as input to the next assignment expression in sequence.

So what does the = operation return? It must return an object, the object it belongs to. We have not seen this yet. We would know how to "return num;" or "return dem;" but we wish to return the entire object. Which object? This object whose operator= function is currently executing. We use a special predefined pointer called **this**.

```
Rational& Rational::operator=(const Rational& r){
   num = r.num;
   den = r.den;
   return *this;        //return a reference to the object that owns
}                       // this function
```

Notice that this is not the same as the copy constructor. The copy constructor is called by an initialization statement and *creates* an object. An assignment calls the assignment operator and operates on objects that are already declared.

The complete program is:

```
//overloading1.cpp
//Rational class w/ overloaded = operator
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   void print();
   Rational& operator= (const Rational&);
private:
   int num, den;
   int gcd (int j, int k) {if (k==0) return j; return gcd(k, j%k);}
   void reduce () {int g = gcd(num, den); num /= g; den /= g;}
};

int main(){
   Rational x(100,360);
   Rational y(x);
   Rational z, w;
   cout << "x=  "; x.print();
   cout << "\ny=  "; y.print();

   cout << "\nz=  "; z.print();
   w = z = y;
   cout << "\nz=  "; z.print();
   cout << "\nw=  "; w.print();

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
```

```
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

void Rational::print(){
   cout << num << '/' << den;
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}
```

Output:

```
x= 5/18
y= 5/18
z= 0/1
z= 5/18
w= 5/18
```

Arithmetic Operators

From Fraction class above:

```
//multiply 2 fractions
fraction fraction::multiply(fraction f); {
fraction temp(num*f.num, denom*f.denom);
return temp;
}
```

We could simply change the name of the function so that we are actually overloading the * operator:

```
//multiply 2 fractions
fraction fraction::operator*(fraction f); {
fraction temp(num*f.num, denom*f.denom);
return temp;
}
```

However, typically, when we wish to add (+) or multiply (*) two objects, neither object really owns the operator+ function or the operator* function. For instance if x and y are both declared as objects of the Rational class, the expression x*y does not mean "multiply my data members (if "I" am x) by the corresponding data members of y." We just want the system to multiply the corresponding data members of x and y, perhaps to print the results, perhaps to assign the new values to a third object. If we define the overloaded operator* function this way:

```
Rational operator* (const Rational& x, const Rational& y){
   Rational z (x.num * y.num, x.den * y.den);
   return z;
}
```

The objects x and y will not allow the function access to their private data members. Here's where the idea of a ***friend*** function comes in.

```
//overloading2.cpp
//Rational class w/ overloaded * operator in friend function
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   void print();
```

```
   Rational& operator= (const Rational&);
private:
   int num, den;
   int gcd (int j, int k) {if (k==0) return j; return gcd(k, j%k);}
   void reduce () {int g = gcd(num, den); num /= g; den /= g;}
};

int main(){
   Rational x(22,7), y(-3, 8), z;
   z = x;
   cout << "\nz=  "; z.print();
   x = y * z;
   cout << "\nx=  "; x.print();

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

void Rational::print(){
   cout << num << '/' << den;
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}
```

Output:

```
z=  22/7
x=  -33/28
```

Notice that the scope resolution operator :: is not used in the implementation because operator* is not a member function of the Rational class; it is a friend of the class, but not part of it.

Arithmetic Assignment Operators

Overloading arithmetic assignment operators (+=, *=, -=) is almost identical to overloading the assignment operator (not to overloading the arithmetic ops).

```
Rational& Rational::operator*=(const Rational& r){
   num = num * r.num;
   den = den * r.den;
   return *this;
}
```

and here's the whole revised program:

```
//overloading3.cpp
//uses Rational class w/ overloaded *= operator
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   void print();
   Rational& operator= (const Rational&);
   Rational& operator*= (const Rational&);
private:
   int num, den;
   int gcd (int, int);
   void reduce ();
};

int main(){
   Rational x(22,7), y(-3, 8), z(x);
   cout << "\nx=  "; x.print();
   x = x * y;
   cout << "\nx=  "; x.print();
   x = z;
   cout << "\nx=  "; x.print();
   x *= y;
   cout << "\nx=  "; x.print();
   x.reduce(); cout << " x =  "; x.print();

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}
```

```
Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

void Rational::print(){
   cout << num << '/' << den;
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational& Rational::operator*= (const Rational& r){
   num = num * r.num;
   den = den * r.den;
   reduce();
   return *this;
}

Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}

//private functions used to reduce fraction
int Rational::gcd (int j, int k) {
   if (k==0) return j; return gcd(k, j%k);
}
void Rational::reduce () {
   int g = gcd(num, den); num /= g; den /= g;
}
```

Output:

```
x=  22/7
x=  -33/28
x=  22/7
x=  66/-56 x =  -33/28
```

Relational Operators

The relational ops are: < > <= >= == !=. These can all be overloaded, using *friend* functions. Relational operations return type int, with values of either 1 (true) or 0 (false).

Here is the revised Rational class with the == relational operator overloaded in a manner consistent with the way we work with fractions.

```
//overloading4.cpp
//uses Rational class w/ overloaded relational operator added
//Borland C++ 5.02
```

```
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
   friend int operator== (const Rational&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   void print();
   Rational& operator= (const Rational&);
   Rational& operator*= (const Rational&);
private:
   int num, den;
   int gcd (int, int);
   void reduce ();
};

int main(){
   Rational x(22,7), y(-3,8), z(x);
   cout << "\nx=  "; x.print();
   cout << "\ny=  "; y.print();
   cout << "\nz=  "; z.print();
   cout << "\nx and y are ";
   if (x==y) cout << "equal";
      else cout << "not equal";
   cout << "\nx and z are ";
   if (x==z) cout << "equal";
      else cout << "not equal";

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

void Rational::print(){
   cout << num << '/' << den;
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational& Rational::operator*= (const Rational& r){
   num = num * r.num;
```

```
   den = den * r.den;
   return *this;
}

int operator== (const Rational& a, const Rational& b){
   return (a.num * b.den == b.num * a.den);
}

Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}

//private functions used to reduce fraction
int Rational::gcd (int j, int k) {
   if (k==0) return j; return gcd(k, j%k);
}

void Rational::reduce () {
   int g = gcd(num, den); num /= g; den /= g;
}
```

Output:

```
x=  22/7
y=  3/-8
z=  22/7
x and y are not equal
x and z are equal
```

Stream Operators: << and >>

We could overload the stream insertion and extraction ops to customize input or output of data. So instead of the z.print function called in a previous example, we could overload the << op and use cout << z; This also requires us to use friend functions.

The ostream class is defined in the iostream.h header file. If x is a Rational object, cout<< x would invoke a call to the overloaded << operator with two objects as arguments: cout and x. The return is cout so the statement can be chained, e.g., cout << "x= " << x;

Program:

```
//overloading5.cpp
//uses Rational class w/ overloaded output operator added
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
```

```
   friend int operator== (const Rational&, const Rational&);
   friend ostream& operator<< (ostream&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   //   void print();  no longer needed
   Rational& operator= (const Rational&);
   Rational& operator*= (const Rational&);
private:
   int num, den;
   int gcd (int, int);
   void reduce ();
};

int main(){
   Rational x(22,7), y;
   cout << "x =  " << x << " and y =  " << y << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational& Rational::operator*= (const Rational& r){
   num = num * r.num;
   den = den * r.den;
   return *this;
}

int operator== (const Rational& a, const Rational& b){
   return (a.num * b.den == b.num * a.den);
}

Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}

ostream& operator<< (ostream& out, const Rational& r){
   return out << r.num << '/' << r.den;
}

//private functions used to reduce fraction
int Rational::gcd (int j, int k) {
```

```
   if (k==0) return j; return gcd(k, j%k);
}

void Rational::reduce () {
   int g = gcd(num, den); num /= g; den /= g;
}
```

Output:

```
x =  22/7 and y =  0/1
```

The input >> operator can be overloaded in a similar way, e.g.:

```
friend istream& operator >> (istream& in, Rational r){
   cout << "numerator = "; in >> r.num;
   cout << "denominator = "; in >> r.den;
   r.reduce();
   return in;
}
```

Increment and Decrement Operators

These are *unary* operators, and we must distinguish between prefix (++x) and postfix (x++).

What does it mean to increment a fraction? For ex.,

$$\frac{23}{7} + 1 = \frac{23+7}{7} = \frac{\text{num} + \text{den}}{7}$$

The pre-increment and post-increment functions look exactly the same: same function name and same number and type of arguments. In order to distinguish the two forms, an extra (unnamed) dummy parameter is included in the post-increment form.

```
//overloading7.cpp
//uses Rational class w/ overloaded ++ operators added
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
   friend int operator== (const Rational&, const Rational&);
   friend ostream& operator<< (ostream&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   Rational& operator= (const Rational&);
```

```
   Rational& operator*= (const Rational&);
   Rational operator++();           //pre-increment
   Rational operator++(int);        //post-increment
private:
   int num, den;
   int gcd (int, int);
   void reduce ();
};

int main(){
   Rational x(23,7), y, z;
   cout << "x =  " << x << "  y = " << y << endl;
   y = x++;
   cout << "x =  " << x << "  y = " << y << endl;
   cout << "\nx =  " << x << "  z = " << z << endl;
   z = ++y;
   cout << "x =  " << x << "  z = " << z << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational& Rational::operator*= (const Rational& r){
   num = num * r.num;
   den = den * r.den;
   return *this;
}

Rational Rational::operator++(){
   num +=den;
   return *this;
}

Rational Rational::operator++(int){
   Rational temp = *this;
   num +=den;
   return temp;
}

int operator== (const Rational& a, const Rational& b){
   return (a.num * b.den == b.num * a.den);
}
```

```
Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}

ostream& operator<< (ostream& out, const Rational& r){
   return out << r.num << '/' << r.den;
}

//private functions used to reduce fraction
int Rational::gcd (int j, int k) {
   if (k==0) return j; return gcd(k, j%k);
}

void Rational::reduce () {
   int g = gcd(num, den); num /= g; den /= g;
}
```

Output:

```
x =  23/7  y = 0/1
x =  30/7  y = 23/7

x =  30/7  z = 0/1
x =  30/7  z = 30/7
```

Conversion Operators

We want type conversion to and from classes to look like any other ordinary type conversion. The syntax for overloading the conversion operator is slightly different from what we have had before.

```
//overloading6.cpp
//uses Rational class w/ overloaded conversion operator added
//Borland C++ 5.02
//Project Target Type: Application - Target Model: Console
//modified from Hubbard

#include <iostream>

class Rational {
   friend Rational operator* (const Rational&, const Rational&);
   friend int operator== (const Rational&, const Rational&);
   friend ostream& operator<< (ostream&, const Rational&);
public:
   Rational (int=0, int=1);
   Rational (const Rational&);
   Rational& operator= (const Rational&);
   Rational& operator*= (const Rational&);
   operator double() const;         //const added for use on
private:                            // const Rationals
   int num, den;
```

```
   int gcd (int, int);
   void reduce ();
};

int main(){
   Rational x(-5,8);
   const Rational PI(22,7);
   cout << "x =  " << x << " = " << double(x) << endl;
   cout << "PI =  " << PI << " = " << double(PI) << endl;

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
   return 0;
}

Rational::Rational (int n, int d) : num (n), den (d){
   reduce();
}

Rational::Rational (const Rational& r) : num(r.num), den(r.den){
}

Rational& Rational::operator= (const Rational& r){
   num = r.num;
   den = r.den;
   return *this;
}

Rational& Rational::operator*= (const Rational& r){
   num = num * r.num;
   den = den * r.den;
   return *this;
}

Rational::operator double () const{
   return double (num)/den;
}

int operator== (const Rational& a, const Rational& b){
   return (a.num * b.den == b.num * a.den);
}

Rational operator* (const Rational& a, const Rational& b){
   Rational r(a.num*b.num, a.den * b.den);
   return r;
}

ostream& operator<< (ostream& out, const Rational& r){
   return out << r.num << '/' << r.den;
}

//private functions used to reduce fraction
int Rational::gcd (int j, int k) {
   if (k==0) return j; return gcd(k, j%k);
}

void Rational::reduce () {
```

```
   int g = gcd(num, den); num /= g; den /= g;
}
```

Output:

```
x =  -5/8 = -0.625
PI =  22/7 = 3.14286
```

Subscript Operator []

The overloaded subscript operator can be used as an access function , providing public access to private data:
e.g,

```
int& Rational::operator[] (int i){
if (i==1) return num;
if (i==2) return den;
}
```

then, in main()
`cout << x[1];` would display the num of x.

Constraints on operator overloading:

What operators can't be overloaded?
. :: ?: sizeof

We can't change the precedence of an operator by overloading.

We can't change the associativity of an operator by overloading.

We can't change the "arity" of an operator by overloading (unary, binary, etc.)

We can't create new operators (e.g., **).

We can't change the way an operator works on objects of built-in types. Operator overloading only works on objects of user-defined (or, rather, programmer-defined) types.