Operator Overloading

Operator Overloading

- C++ philosophy is to make all types "equal"
- Capabilities attached to intrinsic types should be available to user-defined types
  - Use in different memory segments
  - Ability to define operators for these new types
- User defined types should have "standard" operators implemented on them
  - Where it makes sense…

Example – Fraction class

- Declaration

```
class Fraction
{
private:
   int numerator;
   int denominator;
public:
   Fraction();
   Fraction( int num, int denom );
   void Show();
};
```

Default constructor and a constructor that takes the numerator and denominator.

'Show' member function displays the fraction on cout.

Example – Fraction class

- Basic implementation

```
Fraction::Fraction()
{
   // Default to 0/1
   numerator = 0;
   denominator = 1;
}

Fraction::Fraction( int num, int denom )
{
   numerator = num;
   denominator = denom;
}

void Fraction::Show()
{
    cout << numerator << "/" << denominator;
}
```

Adding an Operator Function

- Defined using naming convention
  - *operator* followed by operator character
  - Optional white space for readability
- Example: Addition
  - Define function taking a single parameter
  - Remember implicit *this* argument used for first (left) operand

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

C++ calls the operator+ member function of f1, which adds the members of f2 to f1's own members, returning the resulting Fraction object

```
Fraction Fraction::operator + ( Fraction & rightArg )
{ ...
```

Implementing the Addition Operator

```
Fraction Fraction::operator+( Fraction & rightArg )
{
   if ( denominator == rightArg.denominator )
      return Fraction( numerator + rightArg.numerator,
                       denominator );
   else
      return Fraction( numerator * rightArg.denominator +
                       rightArg.numerator * denominator,
                       denominator * rightArg.denominator );
}
...
Fraction f1( 2, 3 ), f2( 3, 4 ), res;
res = f1 + f2;
res.Show();
```

The function 'belongs' to the left operand and takes the right operand as its argument, returning the result

The + operator can now be used conventionally

Operator Overloading Rules

- Cannot overload operators of intrinsic types
  - At least one operand must be a user-defined type
- Only "built in" operators may be overloaded
  - Additionally ::, ?: and .* cannot be overloaded
- No automatic association of assignment operators with basic operations
  - E.g. += not automatically interpreted as + followed by =
- Precedence and associativity cannot be altered
- Commutativity is not automatically available

Mixing Types

- Operands need not be of the same type

```
Fraction Fraction::operator+( int rightArg )
{
  return Fraction( numerator + rightArg * denominator,
                   denominator );
}
...
Fraction f1( 2, 3 ), res;
res = f1 + 3;
```

An operator+ function that adds an integer (the right operand) to the current Fraction object (the left operand). It adds the integer as a whole number

Operand Types

- Addition implemented as a member function of the Fraction class
  - Requires left operand to be of type Fraction
  - Addition should be commutative (symmetric)

```
Fraction f1( 2,3 ), res;
res = 3 + f1;
```

A member function cannot be used as the left operand is not of a class type

- This can be done by implementing the operator as a free (global) function
  - Requires to be friend, since access to internal state is required

Operator as a Free Function

- Requires to be friend of the class

```
class Fraction
{
 ...
friend Fraction operator+( int leftArg, Fraction & rightArg );
};

Fraction operator+( int leftArg, Fraction & rightArg )
{
  return Fraction( leftArg * rightArg.denominator
                   + rightArg.numerator,
                   rightArg.denominator );
}
```

Adds an int to a Fraction and returns a Fraction

Unary Operators

- Most operators available for overload are binary
- Some operators are unary
  - !, ~, ++, --
- Some can be unary or binary
  - +, -, *, &
- Unary operators implemented as no-arg member functions
  - Or single arg free functions

Unary Operator Example

- ++ for the Fraction class
  - Pre-increment version

```
Fraction Fraction::operator++()
{
   numerator++;
   return *this;
}
```

Returns the value of the current object

```
Fraction f1( 2,3 ), res;
res = ++f1;
res.Show();
cout << "\n";
f1.Show();
cout << "\n";
res = f1++;
res.Show();
cout << "\n";
f1.Show();
```

Displays:
3/3
3/3
4/3
4/3

Gives a compiler warning, postfix operator not available, using prefix operator instead

Postfix Unary Operator

- Implemented as member function with a single int argument
  - Ignored, required to support overloading to different functions

```
Fraction Fraction::operator++( int dummy )
{
    Fraction ret( numerator, denominator );
    numerator++;
    return ret;
}
```

Returns the value before the increment

```
Fraction f1( 2,3 ), res;
res = ++f1;
res.Show();
cout << "\n";
f1.Show();
cout << "\n";
res = f1++;
res.Show();
cout << "\n";
f1.Show();
```

Displays:
3/3
3/3
3/3
4/3

Type Conversion Operators

- Recall that constructors can serve a type conversion role
  - From a type into the required type
- It is also possible to convert *from* a type into another type
  - Requires definition of type conversion operators
- Example, to support conversion of Fraction into double

```
double d;
Fraction f1( 2, 3 );
d = f1;
```

Gives a compiler error as there is no implicit conversion between these types

Type Conversion Operators

- Add conversion operator to Fraction class
  - To convert to double

```
Fraction::operator double() const
{
    return static_cast<double>( numerator ) /
           static_cast<double>( denominator );
}
double d;
Fraction f1( 2, 3 );
d = f1;
```

The conversion function must not have arguments or a return type

The compiler will automatically use the conversion function when required.

- Note that implicit type conversions must be carried out in a single step
  - "transitive" conversions not supported

Copy Constructor and Assignment Operator

- Copy constructor used to build a copy of an existing object
  - Default copy constructor is provided
  - Default assignment operator also provided

```
objectA = objectB;          // Simple assignment, objects already exist
ClassX objectC( objectA );  // Initialization using copy constructor
ClassX objectD = objectA;   // Initialization using copy constructor
```

- Perform "shallow copy"
  - Incorrect if the class supports members accessed by pointer

Copy, Assignment and Pointers

- For class containing pointer members

```
class UsesPointers
{
private:
        Job * pJob;
public:
        UsesPointers();
        ~UsesPointers();
        int GetCost();
};
```

The Job pointed to is allocated in the constructor and freed in the destructor

```
UsesPointers::UsesPointers()
{
    pJob = new Job();
}

UsesPointers::~UsesPointers()
{
    delete pJob;
}
int UsesPointers::GetCost()
{
    return pJob->GetAmount();
}
```

Copy, Assignment and Pointers

- Using the default copy/assignment results in errors

```
UsesPointers * pA;
UsesPointers B;

pA = new UsesPointers;

B = *pA;

delete pA;

int cost = B.GetCost();
```

pA is a pointer to an object, B is an object

Creates an object, pointed to by pA

Copies the object, pointed to by pA to the Object B. C++ will do a member by member copy, including a copy of the pointer pJob

Deletes the object pointed to by pA, including the Job object pointed to by pJob in both objects

Get cost uses the Job object pointed to by pJob but this has been deleted in pA's destructor

Implementing Copy Ctor and Assignment

- Copy constructor is used for intialisation
  - May require deep copy of members
  - No return value
- operator= is used for assignment
  - May require deep copy of members
  - Returns a value
  - May require memory manipulation as existing member is being replaced
  - Should check for, and deal with self-assignment

```
object1 = object1;
```

or more typically:

```
pObject = &object1;
...
object1 = *pObject;
```

Self assignment can cause terminal recursion if not protected against.
The operator= function should compare the address of the assignment variable with the *this* pointer, and do nothing more than returning **this* if they are the same

Adding the Required Functions

```
class UsesPointers
{
private:
    Job * pJob;
public:
    UsesPointers();
    UsesPointers( const UsesPointers & other );
    ~UsesPointers();
    UsesPointers & operator= ( const UsesPointers & other );
    int GetCost();
};
```

The copy constructor and assignment operator both take a reference to an object of the same type

Implementations

- Copy Constructor
  - Requires to create new Job member, which is a copy of Job from input object
  - Default copy constructor can be used for Job, since no pointers

```
UsesPointers::UsesPointers( const UsesPointers & other )
{
        pJob = new Job( *other.pJob );
}
```

The default copy constructor for Job takes an argument which is a reference to the same type

Implementations

- Assignment operator
  - Include check for self-assignment
- Return reference to the current object to support chaining
  - As in basic assignment operator

```
UsesPointers & UsesPointers::operator= ( const UsesPointers & other )
{
    if ( &other != this )
    {
        delete pJob;
        pJob = new Job( *other.pJob );
    }
    return *this;
}
```

Check for self-assignment

Deletes the existing job and replaces it with a copy of the one passed

Other Operators

- Overloading function call operator () allows for construction of objects that behave like functions
  - "Functor"
- Overloading indexing operator [] supports arbitrary indexing in collection types
  - E.g. Map lookup
- Overloading new and delete allow custom handling of dynamic memory
  - Custom heap allocation and management strategies