Introduction

This article talks about the operator overloading in C#. What are the various types of operators that can be overloaded. We will create a small dummy application to see how can we overload some basic operators.

Background

In an object oriented programming language like C#, operator overloading provides a much more natural way of implementing the operations on custom types. Suppose that we have a class created for Complex number and we want to perform all the arithmetic operations on this type. One way to do this is by having functions like Add, Subtract inside the class and have the functionality. Another way is to actually have the overloaded version of operators to act on this type.

Operator overloading provides a much natural abstraction for the types. When we think about possible operation on some data type we can think of binary operators, unary operators, relational operators and perhaps some conversion operations to and from the basic types. In C# achieving all this is possible using operator overloading.

Using the code

Before looking at the implementation details, lets see what are the conventions that need to be followed if we want to overload an operator.

* The operator function should be a member function of the containing type.
* The operator function must be static.
* The operator function must have the keyword operator followed by the operator to be overridden.
* The arguments of the function are the operands.
* The return value of the function is the result of the operation.

So to understand the above mentioned points more clearly we can visualize the operators getting invoked as function calls (below code will not compile, it is only for understanding purpose).

class A

{

public static A operator+(A a1, A a2)

{

A temp;

*// perform actual addition*

return temp;

}

}

*// so the operations like*

A a1 = new A();

A a2 = new A();

A result = a1 + a2;

*// This can actually be visialized as:*

A result = A.+(a1, a2);

*// Note: Only for understnading, this wont even compile*

So following the above guidelines, lets us see how we can implement operator overloading. We will create a small class Rational representing a rational number and will implement some basic overloaded operators in it.

class Rational

{

*// Private vairables to hold the data*

int numerator;

int denominator;

*// Constructor*

public Rational(int num, int den)

{

numerator = num;

denominator = den;

}

public double Value

{

get

{

return ((double)numerator) / denominator;

}

}

}

**Overloading the Binary operators**

Binary operators function will take 2 arguments and return a new object of the Containing type. Let us try to implement the binary operator + for our Rational class.

*// Let us overload the Binary operators*

public static Rational operator +(Rational rational1, Rational rational2)

{

int resultingDenominator = rational1.denominator \* rational2.denominator;

int resultingNumerator = rational1.numerator \* rational2.denominator +

rational1.denominator \* rational2.numerator;

return new Rational(resultingNumerator, resultingDenominator);

}

Once we have the binary operator overloaded we can use it simply as

Rational r1 = new Rational(3, 2);

Rational r2 = new Rational(2, 3);

Rational result = null;

*// Testing + operator on types*

result = r1 + r2;

The important thing to note here is that once we have the binary operator + implemented the compound assignment operator for that operator i.e. += in this case also gets implemented in terms of this operator i.e. + in this case.

The other possibility could be that we may want to have mix mode arithmetic with our type i.e. we may want to have the arithmetic operations to work with our Rational type and some primitive type. So if we want to have the possibility of having Rational + int then we will have to implement an overloaded function for that too.

*// Binary operators for Mix mode arithmetics (Only for + and with int)*

public static Rational operator +(Rational rational1, int num)

{

return new Rational(rational1.numerator +(rational1.denominator \* num), rational1.denominator);

}

public static Rational operator +(int num, Rational rational1)

{

*//lets implement this in terms of other*

return rational1 + num;

}

*//Usage*

static void Main(string[] args)

{

*// Testing + operator in mix mode*

result = r1 + 2;

Console.WriteLine(result.Value.ToString());

*// Testing + operator in mix mode*

result = 3 + r1;

Console.WriteLine(result.Value.ToString());

}

**Overloading the Unary operators**

To overload the unary operators, we need a function taking only one argument of the containing type. The important thing in case of overloading unary operators is that we should not create a new object but instead change the value of the object being passed to us and return it. Let us implement the unary ++ for our Rational type now.

*// Let us overload the unary operator ++ now (-- will be on same lines)*

public static Rational operator ++(Rational rational1)

{

rational1 += 1;

return rational1;

}

*//Usage*

static void Main(string[] args)

{

*// Post increment ++*

result = r1++;

Console.WriteLine(result.Value.ToString());

*// Pre increment ++*

result = ++r1;

Console.WriteLine(result.Value.ToString());

}

The thing to notice in the above code snippet is that we overloaded ++ and C# internally took care of using it in pre-increment and post-increment scenario. So unlike C++ we need not implement the separate versions for prefix and post-fix unary operators.

**Overloading the Relational operators**

Relational operators like < and > can also be overloaded as functions taking two arguments. The important thing to note is that we need to overload relational operators in pairs i.e. if I am overloading < operator than I will have to overload > operator too. Same is true for (<=, >=) and (==, !=) operators. So let us go ahead and implement the relational operators in our Rational class.

*// Lets overload the relational operator*

*// < and > comes in pair, if we define one we need to define other too.*

*// Same is applicable for <= and >=. they also come in pair.*

public static bool operator < (Rational rational1, Rational rational2)

{

return rational1.Value < rational2.Value;

}

public static bool operator >(Rational rational1, Rational rational2)

{

return rational1.Value > rational2.Value;

}

*//Usage*

static void Main(string[] args)

{

*// test relational operator*

Console.WriteLine(r1 > r2);

}

Other important thing to note is while implementing the equality i.e. == operator. If we are overloading ==operator then we need to implement != operator too(as discussed above). Also, we need to override theEquals and GetHashCode functions so that if our object returns true for == then it should return true for Equals function too and should return the same value from GetHashCode().

*// Let is now override the equality operator*

*// == and != comes in pair, if we define one we need to define other too.*

public static bool operator ==(Rational rational1, Rational rational2)

{

return rational1.Value == rational2.Value;

}

public static bool operator !=(Rational rational1, Rational rational2)

{

return rational1.Value != rational2.Value;

}

public override bool Equals(object obj)

{

Rational r = obj as Rational;

if (r != null)

{

return r == this;

}

return false;

}

public override int GetHashCode()

{

return Value.GetHashCode();

}

*//Usage*

static void Main(string[] args)

{

*// test euqlity*

Console.WriteLine(r1 == r2);

}

**Overloading the Conversion operators**

We might also need to implement conversion operators sometimes so that our type can safely be converted to and from other types. We can define the conversion operators as implicit or explicit. In case of implicitconversion the user will not have to explicitly type cast our type to target type and our conversion operation will work. In case of explicit conversion the user will have to explicitly cast our type to target type to invoke our conversion operation. If the casting is not performed it will give a compile time error.

Let us go ahead and define the conversion operation for our Rational type. We will implicitly convert integertypes to Rational type but we will keep the conversion from Rational to double explicit.

*// Finally let us have some conversion operators*

public static implicit operator Rational(int i)

{

*// since the rational equivalant of an int has 1 as denominator*

Rational rational = new Rational(i, 1);

return rational;

}

public static explicit operator double(Rational r)

{

double result = ((double)r.numerator) / r.denominator;

return result;

}

*//Usage*

static void Main(string[] args)

{

*// implicit convertion from int to rational*

Rational ri = 3;

*//explicit conversion from rational to double*

double value = (double)ri;

}

Now We have some basic operators overloaded/implemented for this Rational class. We have implemented some of the operators in each category of operators, rest of the operators can be overloaded on same lines.