UCSD Visual C#.NET Programming II

Description: Working with Stacks and Queues

Setup

Create a new Console Application project called Lab1.

Background

A popular technique for representing and calculating algebraic expressions is to use a syntax known as postfix notation (also known as Reverse Polish Notation). Unlike the more recognizable infix notation where operators are placed between the operands, postfix notation places the operators after the operands.

Examples:

Infix Notation	Postfix Notation
1 + 2	1 2 +
3 + 6 * 4	3 6 4 * +
1 + 6 / 3 + 4 * 2	1 6 3 / + 4 2 * +
3 * (8 - 4)	3 8 4 - *
(1 / 2) * (8 - 4)	1 2 / 8 4 - *

A question that might cross your mind is "why would we do this"? With postfix notation, it is incredibly simple to calculate the expression. Notice in the last example that the parentheses are eliminated from the postfix expression? That is because postfix notation does not need parentheses to correctly calculate the answer. Also notice that the numbers are always in the same order; however, the symbols are repositioned. This is also another curious aspect of postfix notation. In addition to standard binary operators, functions can also be expressed in postfix notation, but that is beyond the scope of this lab. We'll look at postfix calculation later...

Converting to Postfix Notation

To convert from infix to postfix notation we need to use either recursive function calls or stacks and queues. The stack and queue method tends to be easier to understand, even if the results are the same.

The algorithm for converting to postfix is commonly known as the "railroad shunting yard" algorithm. First, the expression is tokenized into values and symbols. Then, using a stack and a queue, the algorithm parses the expression from left-to-right.

Step A

Create a queue called **Q**

Create a stack called **S** – only operator/symbol tokens will ever be on this stack

Convert expression to tokens (this can also be done *on-the-fly*)

Step B – Read the next token

If the token is a number then add it to **Q**

Else if the token is an operator, we'll call it **o1**, (+ - * / % ^) then:

While the top of **S** contains an operator (we'll call **o2**) and either:

o1 is left-associative and its precedence ≤ **o2**'s precedence

OR

o1 is right-associative and its precedence < **o2**'s precedence

Pop **o2** from **S** and append to **Q**

Push **o1** onto **S**

Else if the token is a left parenthesis, push it onto **S**

Else if the token is a right parenthesis then:

While the top of **S** is not a left parenthesis, pop the top operator off **S** and add it to **Q**

Pop the left parenthesis off **S** and discard (no parentheses are ever in **Q**)

If **S** is emptied before encountering a left parenthesis then there was a mismatch and the expression is invalid

Repeat Step B until all tokens have been processed

Step C – Once all tokens have been read, pop all remaining values one-at-a-time from **S** and append to **Q**

Note: If a left parenthesis is encountered there are mismatched parentheses

Q contains the postfix expression

This algorithm may seem complicated, but we'll break it down into manageable tasks.

Implementing the Postfix Conversion Algorithm

Tokenization

One of the first tasks in converting infix to postfix is to tokenize the input expression. This basically means identifying every number and symbol as a separate entity known as a token.

So, the expression "5 + 3" contains three tokens, a number **5**, a symbol **+** and another number **3**.

To manage these tokens we will add a class to our project called *Token*

Token

Token is an **abstract** class that actually contains no code – we just want a common parent for all of the specific token types

```
public abstract class Token
{
}
```

DoubleToken : Token

Next, we will create a Token derivative that will store a **double** number. Call this class *DoubleToken*.

```
public class DoubleToken : Token
{
}
```

Within this *DoubleToken* class we need to add an auto property called *Value* which is of type **double**.

```
public double Value { get; set; }
```

Next, add a constructor that accepts a **double** and assign that value to the *Value* property.

Finally, add a *ToString()* override that returns the *Value* property converted to a **string**.

SymbolToken : Token

The next *Token* derivative to add is called *SymbolToken* and it is considerably more complex than *DoubleToken* (make sure you inherit from *Token*)

SymbolToken contains an **enum** called *SymbolsEnum* where each value represents one of the possible symbol tokens:

```
public enum SymbolsEnum
{
    Unknown = 0,
    Add,
```

```
        Subtract,
        Multiply,
        Divide,
        Modulus,
        Power,
        LParen,
        RParen
    }
```

Next, create a **private** *SymbolsEnum* field in *SymbolToken* called *m_Symbol* and set it to *SymbolsEnum.Unknown*.

Wrap this *m_Symbol* field with a property called *Symbol*. The **get** accessor simply returns *m_Symbol*. The **set** accessor should validate that the passed in value is defined within *SymbolsEnum*.

```
    public SymbolsEnum Symbol
    {
        get
        {
            return m_Symbol;
        }
        set
        {
            if (Enum.IsDefined(typeof(SymbolsEnum), value))
            {
                m_Symbol = value;
            }
            else
            {
                throw new Exception("Unknown symbol.");
            }
        }
    }
```

As you may have noticed in the algorithm, we need to determine the precedence of an operator. To do this, we will create an **int** property called *Precedence* which only has a **get** accessor. The value returned from this property is determined by the contents of the *Symbol* property. Use a **switch** statement to return the correct value based on the following chart:

Operator	Precedence
Add, Subtract	1
Multiply, Divide, Modulus	2
Power	3
Anything else (default)	0

The next property we will add is for quickly determining if the symbol contained with a *SymbolToken* object is an operator. We will call this property *IsOperator* and it will return a **bool** and also only have a **get** accessor. The return value is **true** if the value of *Precedence* is greater than 0. Otherwise, the return value is **false**. This makes sense as the *Unknown*, *LParen* and *RParen* values of *SymbolsEnum* will cause *Precednece* to return 0.

In the algorithm you may have also noticed the terms "left-associative" and "right-associative". The only time this is important is if two operators of the same precedence appear consecutively within an infix expression. If an operator is left-associative it will perform the left operator calculation first, whereas a right-associative operator will favor calculating the right operator first. To provide this information to the parser, we will create two **bool** properties called *IsLeftAssociative* and *IsRightAssociative*. Again, these will only contain **get** accessors. Since *Power* is the only right-associative operator we support, the code is very straightforward: *IsLeftAssociative* returns **true** only if *IsOperator* is **true** AND *Symbol* does not equal *Power*. IsRightAssociative returns **true** only if *Symbol* equals *Power*.

We now need a constructor for the class. It needs to accept a *SymbolsEnum* value and assign it to the *Symbol* property.

Next we need an alternative way of creating a new instance of this class passing in a **char**. We could, of course, create another constructor for this task. However, I'd like to demonstrate the "factory" model for creating an object. In this case we need a **public static** method that accepts a **char** and returns a *SymbolToken*. This factory model is very commonly used in object-oriented programming, so it is a good idea to start practicing with the concept. Here is the code for the *FromChar* factory method:

```
public static SymbolToken FromChar(char ch)
{
    switch (ch)
    {
        case '+':
            return new SymbolToken(SymbolsEnum.Add);
```

```
            case '-':
                return new SymbolToken(SymbolsEnum.Subtract);
            case '*':
                return new SymbolToken(SymbolsEnum.Multiply);
            case '/':
                return new SymbolToken(SymbolsEnum.Divide);
            case '%':
                return new SymbolToken(SymbolsEnum.Modulus);
            case '(':
                return new SymbolToken(SymbolsEnum.LParen);
            case ')':
                return new SymbolToken(SymbolsEnum.RParen);
            case '^':
                return new SymbolToken(SymbolsEnum.Power);
            default:
                return new SymbolToken(SymbolsEnum.Unknown);
        }
    }
```

Notice that the *FromChar* method makes use of the constructor we created. Again, this is a common technique for generating objects.

Finally, add a *ToString()* override that returns the **string** representation of the stored *Symbol*. Basically, it will do just the opposite of the *FromChar* method – it will **switch** upon *Symbol* and return the corresponding **string**.

Make sure your code compiles before continuing. Then add some test code to Program.Main to ensure your code is functioning properly:

```
DoubleToken num1 = new DoubleToken(123.45);
Console.WriteLine(num1);
SymbolToken sym1 = new SymbolToken(SymbolToken.SymbolsEnum.Power);
Console.WriteLine("Symbol: {0}, IsOp: {1}, Precedence: {2}, LtAssoc: {3}, RtAssoc: {4}",
    sym1, sym1.IsOperator, sym1.Precedence, sym1.IsLeftAssociative, sym1.IsRightAssociative);
sym1 = SymbolToken.FromChar('*');
Console.WriteLine("Symbol: {0}, IsOp: {1}, Precedence: {2}, LtAssoc: {3}, RtAssoc: {4}",
    sym1, sym1.IsOperator, sym1.Precedence, sym1.IsLeftAssociative, sym1.IsRightAssociative);
```

Output:

```
123.45
Symbol: ^, IsOp: True, Precedence: 3, LtAssoc: False, RtAssoc: True
Symbol: *, IsOp: True, Precedence: 2, LtAssoc: True, RtAssoc: False
```

SymbolTokenStack : Stack<SymbolToken>

Next, create a new class called *SymbolTokenStack* that derives from **Stack**<*SymbolToken*>. This class does not actually add to the base functionality of **Stack**<T>. Rather, we are using inheritance to demonstrate how you can make a specific version of a generic class that we can create instances of in other parts of the code.

```
public class SymbolTokenStack : Stack<SymbolToken>
{
}
```

PostfixQueue : Queue<Token>

Now, we are ready to create the class that will do the majority of the processing. This class, called *PostfixQueue*, is a derivative of **Queue**<*Token*>. As implied, this class is a **Queue**, which means items are processed in First-In-First-Out order, using the Enqueue and Dequeue methods.

The constructor for *PostfixQueue* accepts a **string** called *input*. Inside the constructor make a call to a method called *Tokenize*, passing *input* as a parameter. We'll define the *Tokenize* method next.

Tokenize is the method that will convert an infix expression to postfix using the algorithm identified above. This method is **public**, accepts a **string** called *input* and returns an **int** which is the number of tokens found. I made some language-specific changes to the algorithm to make it more manageable.

Step 0 - Setup

Clear the queue by calling *Clear()*. Remember, this method is in a class that IS a **Queue**<*Token*> object, so it automatically has all of the methods of **Queue**<T>.

Create a *StringBuilder* called *number*.

Create a *SymbolTokenStack* called *stack*.

Remove all spaces from *input* using the *Replace* function

```
Clear();
StringBuilder number = new StringBuilder();
SymbolTokenStack stack = new SymbolTokenStack();

// Trim all whitespace
input = input.Replace(" ", string.Empty);
```

Step 1

Using a **foreach** statement, loop through all the characters in *input*.

```
foreach (char ch in input)
```

Within the **foreach** loop do the following:

Step 1.A

We need to check if the character we are on is of a numeric type. If it is, we don't know how long it will be, so keep adding the digits to the *number* **StringBuilder**. When we encounter something other than a digit we'll know that *number* is complete.

Check if *ch* is one of the following characters: 0 – 9 or '.'. If it is, append *ch* to *number*.

```
if ("0123456789.".Contains(ch))
{
    number.Append(ch);
}
```

Step 1.B

Otherwise, if *ch* is not a numeric character, it must be a symbol. If *number*'s *Length* is not 0, we have to take its contents and convert it to a **double** and store it in a new *DoubleToken*. Do the following:

Step 1.B.1

If *number*'s length > 0 then enqueue a new *DoubleToken* with the contents of *number* converted to a **double**.

```
if (number.Length > 0)
{
    Enqueue(new DoubleToken(double.Parse(number.ToString())));
    number = new StringBuilder();
}
```

Create a *SymbolToken* called symbol using *SymbolToken*'s *FromChar* method, passing in *ch*.

```
SymbolToken symbol = SymbolToken.FromChar(ch);
```

Step 1.B.2

If *symbol* is an operator, we need to check the *stack* for any higher precedence operators before pushing *symbol* onto the *stack*. Add the following code:

```
if (symbol.IsOperator)
{
    while (stack.Count > 0)
    {
        SymbolToken top = stack.Peek();
        if ((symbol.IsLeftAssociative && (symbol.Precedence <= top.Precedence)) ||
```

```
            (symbol.IsRightAssociative && (symbol.Precedence < top.Precedence)))
        {
            Enqueue(stack.Pop());
        }
        else
        {
            break;
        }
    }
    stack.Push(symbol);
}
```

Step 1.B.3

Else if *symbol* is a left parenthesis, push *symbol* directly onto *stack*. We will deal with it when we encounter a right parenthesis:

```
else if (symbol.Symbol == SymbolToken.SymbolsEnum.LParen)
{
    stack.Push(symbol);
}
```

Step 1.B.4

Else if *symbol* is a right parenthesis, we need to pop all of the symbols off of *stack* and enqueue them until we find its matching left parenthesis. If the left parenthesis is not found there were mismatched parentheses in the expression. Add the following code:

```
else if (symbol.Symbol == SymbolToken.SymbolsEnum.RParen)
{
    bool lParenFound = false;
    while (stack.Count > 0)
    {
        SymbolToken top = stack.Peek();
        if (top.Symbol == SymbolToken.SymbolsEnum.LParen)
        {
            // Pop the left parens and exit the loop
            lParenFound = true;
            stack.Pop();
            break;
        }
        else
        {
            Enqueue(stack.Pop());
        }
    }
    if (!lParenFound)
    {
        throw new Exception("Mismatched parentheses");
    }
}
```

End of foreach loop

Step 2

If *number*'s length > 0, then we have the last number of the expression. Enqueue a new *DoubleToken* with the contents of *number* converted to a **double**. *This is after the close brace of the **foreach** loop.*

```
if (number.Length > 0)
{
    Enqueue(new DoubleToken(double.Parse(number.ToString())));
    number = new StringBuilder();
}
```

Step 3

If *stack* is not empty, we have low precedence operators that need to be added to the queue.

While *stack* is not empty, pop a value from *stack*, called *token*. If token is an operator, enqueue token, otherwise throw an exception:

```
while (stack.Count > 0)
{
    SymbolToken token = stack.Pop();
    if (token.IsOperator)
    {
        Enqueue(token);
    }
    else
    {
        throw new Exception("Invalid expression");
    }
}
```

Step 4

Return the number of tokens in the queue (remember, this IS a **Queue**<T> class, so we automatically have a *Count* property):

```
return Count;
```

The complete code for Tokenize can be found in the appendix of this lab.

Next add an override for *ToString()* that outputs the tokens in the queue:

```
public override string ToString()
{
    StringBuilder sb = new StringBuilder();
    foreach (Token token in this)
    {
```

```
            sb.Append(" " + token.ToString());
        }
        return sb.ToString();
    }
```

Now, make sure your code compiles and test the code using the following in Program.Main:

```
PostfixQueue queue = new PostfixQueue("1 + 2 - 3 ^ 4 * 5 / 10 % 3");
Console.WriteLine(queue);
queue = new PostfixQueue("1.7*3.4+3.4/12.5");
Console.WriteLine(queue);
queue = new PostfixQueue("(1 + 4) / 10 + (1.5 * 3) ^ 2");
Console.WriteLine(queue);
queue = new PostfixQueue("7 / (3 * 2 + 4 - 10)");
Console.WriteLine(queue);
```

Output:

```
1 2 + 3 4 ^ 5 * 10 / 3 % -
1.7 3.4 * 3.4 12.5 / +
1 4 + 10 / 1.5 3 * 2 ^ +
7 3 2 * 4 + 10 - /
```

Evaluating the Postfix Expression

Once you have successfully converted infix to postfix, calculation of the expression is surprisingly simple.

When calculating a postfix expression, we once again need a **Stack** and the *PostfixQueue*.

This time, however, the **Stack** (called *calc*) contains only **doubles**, so we will make a simple **Stack**<**double**>.

As we step through the queue containing the tokens we either get a *DoubleToken* or a *SymbolToken*.

Any time a *DoubleToken* is encountered you push the **double** value directly into *calc*.

If a *SymbolToken* is encountered then perform the appropriate operation on the top two values on *calc*. The result is pushed back into the stack.

Once all tokens are processed out of the queue there will be only one value in the stack which contains the result.

PostfixQueue.Calculate

Create a **public** method in *PostfixQueue* called *Calculate* that accepts no parameters and returns a **double**.

Create and initialize a **Stack**<**double**> called *calc*.

Loop through each *Token* in the queue's list:

```
foreach (Token token in this)
```

If *token* is a *DoubleToken*, cast *token* as a *DoubleToken* and Push its *Value* property onto *calc*.

```
if (token is DoubleToken)
{
    calc.Push((token as DoubleToken).Value);
}
```

Otherwise, if *token* is a *SymbolToken*, create a SymbolToken called *symbol* and cast *token* into it. Then, pop two **doubles** off of *calc*, into **doubles** called *op1* and *op2*. Note: if there aren't at least two values in *calc* there is a syntax error with the original postfix expression.

```
else if (token is SymbolToken)
{
    SymbolToken symbol = token as SymbolToken;
    double op1 = calc.Pop();
    double op2 = calc.Pop();
```

Now, using a switch statement on *symbol.Symbol*, perform the appropriate mathematic operation, calculating the result of ***op2 symbol op1***. Notice that you calculate the operands in the opposite order they are popped off the stack.

- *Remember, only the four basic arithmetic operators, modulus and power (see help for Math.Pow) will be switched upon – all parentheses have been culled by the postfix algorithm.*
- *Also, don't worry about dividing by zero – with doubles the answer will be infinity.*

Push the result of the calculation back onto *calc*.

Once all tokens have been processed in the queue, there should be only one **double** left in *calc*. That is the result of the equation. Return that final Pop from *calc*.

You can see a portion of the Calculate code in the appendix of this lab.

Finally, using the same equations from the previous test, calculate and output the results:

```
PostfixQueue queue = new PostfixQueue("1 + 2 - 3 ^ 4 * 5 / 10 % 3");
Console.WriteLine(queue);
Console.WriteLine(queue.Calculate());

queue = new PostfixQueue("1.7*3.4+3.4/12.5");
```

```
Console.WriteLine(queue);
Console.WriteLine(queue.Calculate());

queue = new PostfixQueue("(1 + 4) / 10 + (1.5 * 3) ^ 2");
Console.WriteLine(queue);
Console.WriteLine(queue.Calculate());

queue = new PostfixQueue("7 / (3 * 2 + 4 - 10)");
Console.WriteLine(queue);
Console.WriteLine(queue.Calculate());
```

Output:

```
 1 2 + 3 4 ^ 5 * 10 / 3 % -
1.5
 1.7 3.4 * 3.4 12.5 / +
6.052
 1 4 + 10 / 1.5 3 * 2 ^ +
20.75
 7 3 2 * 4 + 10 - /
Infinity
```

Appendix A – PostfixQueue.Tokenize()

```
/// <summary>
/// Tokenize the input and return the number of tokens found
/// </summary>
/// <param name="input">Input to process</param>
/// <returns>Number of tokens found</returns>
public int Tokenize(string input)
{
    // STEP 0
    Clear();
    StringBuilder number = new StringBuilder();
    SymbolTokenStack stack = new SymbolTokenStack();
    // Trim all whitespace
    input = input.Replace(" ", string.Empty);

    // STEP 1
    foreach (char ch in input)
    {
        // STEP 1.A
        if ("0123456789.".Contains(ch))
        {
            number.Append(ch);
        }
        // STEP 1.B
        else
        {
            // STEP 1.B.1
            // Check if a number needs to be placed in the queue
            if (number.Length > 0)
            {
                Enqueue(new DoubleToken(double.Parse(number.ToString())));
                number = new StringBuilder();
            }

            // Process symbol token
            SymbolToken symbol = SymbolToken.FromChar(ch);

            // STEP 1.B.2
            if (symbol.IsOperator)
            {
                while (stack.Count > 0)
                {
                    SymbolToken top = stack.Peek();
                    if ((symbol.IsLeftAssociative && (symbol.Precedence <= top.Precedence)) ||
                        (symbol.IsRightAssociative && (symbol.Precedence < top.Precedence)))
                    {
                        Enqueue(stack.Pop());
                    }
                    else
                    {
                        break;
                    }
                }
                stack.Push(symbol);
            }
            // STEP 1.B.3
            else if (symbol.Symbol == SymbolToken.SymbolsEnum.LParen)
            {
                stack.Push(symbol);
```

```
            }
            // STEP 1.B.4
            else if (symbol.Symbol == SymbolToken.SymbolsEnum.RParen)
            {
                bool lParenFound = false;
                while (stack.Count > 0)
                {
                    SymbolToken top = stack.Peek();
                    if (top.Symbol == SymbolToken.SymbolsEnum.LParen)
                    {
                        // Pop the left parens and exit the loop
                        lParenFound = true;
                        stack.Pop();
                        break;
                    }
                    else
                    {
                        Enqueue(stack.Pop());
                    }
                }
                if (!lParenFound)
                {
                    throw new Exception("Mismatched parentheses");
                }
            }
        }
    }

    // STEP 2
    // Check if a number needs to be placed in the queue
    if (number.Length > 0)
    {
        Enqueue(new DoubleToken(double.Parse(number.ToString())));
        number = new StringBuilder();
    }

    // STEP 3
    // Enqueue all remaining tokens
    while (stack.Count > 0)
    {
        SymbolToken token = stack.Pop();
        if (token.IsOperator)
        {
            Enqueue(token);
        }
        else
        {
            throw new Exception("Invalid expression");
        }
    }

    // STEP 4
    return Count;
}
```

Appendix A – PostfixQueue.Calculate()

```
/// <summary>
/// Calculates the expression stored in the queue
/// </summary>
/// <returns>Calculated value</returns>
public double Calculate()
{
    Stack<double> calc = new Stack<double>();

    foreach (Token token in this)
    {
        if (token is DoubleToken)
        {
            calc.Push((token as DoubleToken).Value);
        }
        else if (token is SymbolToken)
        {
            SymbolToken symbol = token as SymbolToken;
            if (calc.Count >= 2)
            {
                double op1 = calc.Pop();
                double op2 = calc.Pop();
                switch (symbol.Symbol)
                {
                    case SymbolToken.SymbolsEnum.Add:
                        calc.Push(op2 + op1);
                        break;
                    // Do remaining operations here...                                    }
            }
        }
    }
    if (calc.Count == 1)
    {
        return calc.Pop();
    }
    throw new Exception("Error calculating result");
}
```