

Forum:[Hack Forums](#)

Thread: [C# alternative to StringBuilder, 70% faster and 72% less memory allocation.](#)

Justin Miller - December 5, 2019, 6:26 pm

[github.com/justinamiller/LiteStringBuilder](#)

gigajew - December 6, 2019, 3:18 pm

Cool project but doubt it does any good with pinvoke calls to the win api because windows like wchar strings and will convert them internally before it get passed on

For other purposes pretty cool

Justin Miller - December 11, 2019, 5:45 am

True but the performance is amazing

MavenGradle - December 11, 2019, 10:24 am

This is some constructive criticism that may or may not help you on your development road. [Don't be like this guy.](#)

LiteStringBuilder class:

First of all, the LiteStringBuilder class is incredibly large. I would advise you to split it up into multiple classes, each performing their own actions. There are quite a few imports that hold no purpose, such as:

```
Code using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Versioning;
using System.Text;
using System.Runtime.InteropServices;
```

The following variables are declared using default values and can also be simplified:

```
Code private char[] _buffer = null;
private int _bufferPos = 0;
private int _charsCapacity = 0;
Code private char[] _buffer;
private int _bufferPos;
private int _charsCapacity;
```

Some properties can be simplified. For example the Length getter:

```
Code public int Length
{
    get
    {
        return _bufferPos;
    }
}
```

Can be rewritten as:

```
Code public int Length => _bufferPos;
```

You also have a char[][] that you gave a weird name (s_bool). I would advise you to use names that properly reflect the function or usage of a variable, like you did with the Length property. Also, you can simplify this code:

```
Code private readonly static char[][] s_bool = new char[2][]
```

```

{
    new char[] { 'F', 'a', 'l', 's', 'e' },
    new char[] { 'T', 'r', 'u', 'e' }
};

```

by using something like (C# 8):

```

Code private static readonly char[][] s_bool = {
    new[] { 'F', 'a', 'l', 's', 'e' },
    new[] { 'T', 'r', 'u', 'e' }
};

```

Also notice how I put static before readonly? That's because inconsistent modifier declarations aren't pretty to look at. There are more variables that you declare using inconsistent modifier orders.

In the 'this' return function, you throw an `ArgumentOutOfRangeException` like so:

```

Code if (index > _bufferPos || 0 > index)
{
    throw new ArgumentOutOfRangeException("index");
}

```

I would advise you to use the `nameof` operator instead for better maintainability:

```

Code if (index > _bufferPos || 0 > index)
{
    throw new ArgumentOutOfRangeException(nameof(index));
}

```

More information can be found here:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof>

You also have a lot of these keywords that can be avoided:

Code if (`other.Length != this.Length`)

Code if (`other.Length != Length`)

In the `Set` method you implement a `for` loop. I would advise you to implement a `foreach` loop instead for the sake of readability:

Code for (`int i = 0; i < values.Length; i++`)

```

{
    Append(values[i]);
}
Code foreach (int t in values)
{
    Append(t);
}

```

The following method can also be simplified and made more usable, using a simple trick:

Code `public LiteStringBuilder Append<T>(params T[] values)`

```

{
    if (values != null)
    {
        //Type type = typeof(T);
        //var value = (T)Convert.ChangeType(values[i], type);

        int len = values.Length;
        for (var i = 0; i < len; i++)
        {
            Append(values[i]);
        }
    }
    return this;
}

```

Code public LiteStringBuilder Append<T>(params T[] values)

```
{
    if (values == null) return this;

    foreach (var t in values)
    {
        Append(t);
    }
    return this;
}
```

Did you notice how I did not declare a variable simply to retrieve the length/size of the values array?

In the following Append method, you once again implement a lot of this keywords that can be avoided and simplified using a switch:

Code private void Append<T>(T value)

```
{
    if(value == null)
    {
        //null no need to add
        return;
    }
    if (value is string)
    {
        this.Append(value as string);
    }
}
```

...

Code switch (value)

```
{
    case null:
        //null no need to add
        return;
    case string s:
        this.Append(s);
        break;
    case char _:
        this.Append((char)(object)value);
        break;
    case char[] _:
        this.Append((char[])(object)value);
        break;
    case int _:
        this.Append((int)(object)value);
        break;
}
```

...Regardless of using an if or switch statement, this is incredibly hard to maintain and you should look into better ways to handle this.

The following Append method can also be simplified like so:

Code public LiteStringBuilder Append(bool value)

```
{
    if (value)
    {
        //true
        return Append(s_bool[1]);
    }
}
```

```

    }
    else
    {
        //false
        return Append(s_bool[0]);
    }
}
Code public LiteStringBuilder Append(bool value)
{
    return Append(value ? s_bool[1] : s_bool[0]);
}

```

I find that this greatly improves the readability. Same goes for the next Append method that can be simplified:

```

Code public LiteStringBuilder Append(char[] value)
{
    if (value != null)
    {
        int n = value.Length;
        if (n > 0)
        {
            //EnsureCapacity(n);
            ///Buffer.BlockCopy(value, 0, _buffer, _bufferPos * 2, n * 2);

            //_bufferPos += n;

            EnsureCapacity(n);
            new Span<char>(value).TryCopyTo(new Span<char>(_buffer, _bufferPos, n));
            _bufferPos += n;
        }
    }
    return this;
}

```

```

Code public LiteStringBuilder Append(char[] value)
{
    if (value == null || value.Length == 0) return this;

    EnsureCapacity(value.Length);
    new Span<char>(value).TryCopyTo(new Span<char>(_buffer, _bufferPos, value.Length));
    _bufferPos += value.Length;
    return this;
}

```

The next Append method can also be simplified:

```

Code public LiteStringBuilder Append(object value)
{
    if (value is null)
        return this;

    return Append(value.ToString());
}
Code public LiteStringBuilder Append(object value)
{
    return value is null ? this : Append(value.ToString());
}

```

This Append method can once again be simplified. Take the cast to int for example that's not required:

```
Code public LiteStringBuilder Append(sbyte value)
{
    if (value < 0)
    {
        return Append((ulong)-((int)value), true);
    }
    return Append((ulong)value, false);
}
```

```
Code public LiteStringBuilder Append(sbyte value)
{
```

```
    return value < 0 ? Append((ulong)-value, true) : Append((ulong)value, false);
```

} You perform quite a few unnecessary casts in your code. I'm not going to list them all because I'd be writing this post for another hour or two.

The Replace function is quite another beast that you need to take a look at. I see quite a few exceptions there waiting to happen:

```
Code public LiteStringBuilder Replace(string oldStr, string newStr)
```

```
{
    if (_bufferPos == 0)
        return this;
```

```
...
```

```
    char firstChar = oldStr[0];
```

You never check if oldStr actually has any values at all which opens you up to NullReferenceException cases. There are a couple of others in there that I deliberately did not mention, which I'm sure you can figure out. If not, feel free to let me know and I'll point them out to you.

The following loop is frowned upon

Code do

```
{
    buffer[nbChars--] = _charNumbers[value % 10];
    value /= 10;
} while (value != 0);
```

Why not implement a regular while loop instead? It's way more readable.

Finally, in the GetHashCode method, you use a variable that should be read-only:

Code _bufferPos

There are quite a few other methods that could be simplified but I'm sure you get the picture with the comments that I already gave you. Feel free to let me know if you want me to look further but for now that's all I'm going to point out due to a lack of time.

Utilities class:

The only using statement you need is this one:

Code using System.Runtime.CompilerServices;

In the GetIntLength method, you can simplify the final if statement like so:

Code return n < 100000000000000000000L ? 19 : 20;

In the GetPaddedCapacity method, you can simplify this:

```
Code if (num1 > 1U)
{
```

```
    num1 >>= 1;
    ++num2;
}
return 16 << (num2 + (int)num1);
```

into this:

```
Code if (num1 <= 1U) return 16 << (num2 + (int) num1);
    num1 >>= 1;
    num2++;
    return 16 << (num2 + (int)num1);
```

Notice how I also changed the prefix increment operator to be postfix?

Anyway, that's all I have time for to look into right now. I would really suggest that you refactor and simplify your code because as it stands right now, your code would not be accepted if you committed this into a repository that I control. That's not to say it's not good or does not perform better. Quite simply it means that you have a lot of work to do before your repository would be acceptable in my eyes. Read my comments to understand why that is.

If I can give you another tip, take a look at git-flow. You are working on a master branch that's ahead of the development branch which is extremely weird when it comes to working with git and can cause a lot of confusion for developers that look at your code. More information about git flow can be found here:

<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

Spy™ - December 23, 2019, 11:41 am

does this have a run-time trace that indicates it is better?