C# Operator Overloading, Indexers & StringBuilder - Complete Guide

1. Operator Overloading

Operator overloading allows you to **define custom behavior** for operators (+, -, *, /, ==, etc.) when used with your custom types (classes and structs).

Why Operator Overloading?

Without operator overloading:

```
complex c1 = new complex(3, 4);
complex c2 = new complex(5, 6);

// ❌ This doesn't work without overloading
// complex c3 = c1 + c2;  // Compile error!

// ✅ You'd need a method instead
complex c3 = c1.Add(c2);  // Less intuitive
```

With operator overloading:

```
complex c1 = new complex(3, 4);
complex c2 = new complex(5, 6);

// ✅ Now this works naturally!
complex c3 = c1 + c2;  // Intuitive and clean
```

Syntax for Operator Overloading:

```
public static ReturnType operator OperatorSymbol(ParameterType param1,
ParameterType param2)
{
    // Implementation
    return result;
}
```

Key requirements:

- Must be `public static`

- Must use the `operator` keyword
- Parameters define the operand types
- Return type is the result type

---

2. Binary Operator Overloading

Binary operators work with **two operands** (e.g., a + b).

Addition Operator (+)

Example from code:

```
public static complex operator +(complex a, complex b)
{
    return new complex(a.real + b.real, a.img + b.img);
}
```

How it works:

```
complex c1 = new complex(3, 4);  // 3 + 4i
complex c2 = new complex(5, 6);  // 5 + 6i
complex c3 = c1 + c2;            // 8 + 10i

// Behind the scenes, compiler translates to:
// complex c3 = complex.operator+(c1, c2);
```

Mathematical explanation:

```
Complex number addition:
(a + bi) + (c + di) = (a + c) + (b + d)i

Example:
(3 + 4i) + (5 + 6i) = (3 + 5) + (4 + 6)i = 8 + 10i
```

Overloading with Different Types

Example from code:

```
public static complex operator +(complex a, int b)
{
```

```
    return new complex(a.real + b, a.img);
}
```

Usage:

```
complex c1 = new complex(3, 4);  // 3 + 4i
complex c2 = c1 + 5;             // 8 + 4i (adds 5 to real part only)

// Note: This only works for (complex + int)
// For (int + complex), you need another overload:
public static complex operator +(int a, complex b)
{
    return new complex(a + b.real, b.img);
}
```

Common Binary Operators:

```
// Arithmetic operators
public static complex operator +(complex a, complex b) { }
public static complex operator -(complex a, complex b) { }
public static complex operator *(complex a, complex b) { }
public static complex operator /(complex a, complex b) { }
public static complex operator %(complex a, complex b) { }

// Example: Subtraction
public static complex operator -(complex a, complex b)
{
    return new complex(a.real - b.real, a.img - b.img);
}

// Example: Multiplication
// (a + bi)(c + di) = (ac - bd) + (ad + bc)i
public static complex operator *(complex a, complex b)
{
    return new complex(
        a.real * b.real - a.img * b.img,
        a.real * b.img + a.img * b.real
    );
}
```

---

3. Unary Operator Overloading

Unary operators work with **one operand** (e.g., ++a, -a).

Increment Operator (++)

Example from code:

```
public static complex operator ++(complex a)
{
    a.real++;
    return a;
}
```

Usage:

```
complex c1 = new complex(3, 4);  // 3 + 4i
c1++;                            // 4 + 4i
Console.WriteLine(c1);           // Output: 4+4i
```

Other Unary Operators:

```
// Decrement
public static complex operator --(complex a)
{
    a.real--;
    return a;
}

// Negation
public static complex operator -(complex a)
{
    return new complex(-a.real, -a.img);
}

// Plus (unary +)
public static complex operator +(complex a)
{
    return a;  // Usually returns copy
}

// Logical NOT (!)
public static bool operator !(complex a)
{
    return a.real == 0 && a.img == 0;  // True if complex is zero
}
```

```
// Bitwise NOT (~) - for integer-based types
public static complex operator ~(complex a)
{
    return new complex(~a.real, ~a.img);
}
```

Usage examples:

```
complex c1 = new complex(3, 4);

c1++;           // Increment: 4 + 4i
c1--;           // Decrement: 3 + 4i
complex c2 = -c1;  // Negate: -3 + -4i
bool isZero = !c1; // Check if zero: false
```

---

4. Comparison Operator Overloading

Comparison operators return `bool` values.

Equality Operators (== and !=)

Example from code:

```
public static bool operator ==(complex a, complex b)
{
    return (a.real == b.real && a.img == b.img);
}

public static bool operator !=(complex a, complex b)
{
    return (a.real != b.real || a.img != b.img);
}
```

Important rule: If you overload `==`, you **must** overload `!=` (they come in pairs).

Usage:

```
complex c1 = new complex(3, 4);
complex c2 = new complex(3, 4);
complex c3 = new complex(5, 6);
```

```
if (c1 == c2)  // true (same values)
{
    Console.WriteLine("equal");
}

if (c1 != c3)  // true (different values)
{
    Console.WriteLine("not equal");
}
```

Best Practice: Override Equals() and GetHashCode()

When you overload `==`, you should also override `Equals()` and `GetHashCode()`:

```
public override bool Equals(object obj)
{
    if (obj == null || !(obj is complex))
        return false;

    complex other = (complex)obj;
    return this == other;  // Use our overloaded operator
}

public override int GetHashCode()
{
    return HashCode.Combine(real, img);
}
```

Relational Operators (<, >, <=, >=)

These also come in pairs:

```
// Must overload < and > together
public static bool operator <(complex a, complex b)
{
    // Compare magnitudes: sqrt(real² + img²)
    double magA = Math.Sqrt(a.real * a.real + a.img * a.img);
    double magB = Math.Sqrt(b.real * b.real + b.img * b.img);
    return magA < magB;
}

public static bool operator >(complex a, complex b)
{
    double magA = Math.Sqrt(a.real * a.real + a.img * a.img);
    double magB = Math.Sqrt(b.real * b.real + b.img * b.img);
```

```
    return magA > magB;
}

// Must overload <= and >= together
public static bool operator <=(complex a, complex b)
{
    return !(a > b);
}

public static bool operator >=(complex a, complex b)
{
    return !(a < b);
}
```

---

5. Non-Overloadable Operators

Some operators **cannot** be overloaded:

Cannot Overload:

- **Assignment operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - *Note:* If you overload `+`, then `+=` automatically works
- **Member access**: `.` (dot operator)
- **Array indexing**: `[]` (but you can use indexers - see section below)
- **Conditional**: `&&`, `||` (but you can overload `&` and `|`)
- **Logical NOT**: `!` (can be overloaded for custom logic)
- **Type testing**: `is`, `as`, `typeof`
- **Others**: `=>`, `?:`, `??`, `?.`, etc.

Example from code comment:

```
// non-overloading operator
// += ,-=,*=,/= ,=,.,[],!,&&,||
```

Compound Assignment Operators:

```
complex c1 = new complex(3, 4);
complex c2 = new complex(5, 6);

// ✅ This works automatically if you overload +
c1 += c2;  // Equivalent to: c1 = c1 + c2
```

```
// Compiler automatically translates:
// c1 += c2  →  c1 = c1.operator+(c2)
```

---

6. User-Defined Type Conversions (Casting)

You can define how your custom type converts to/from other types.

Implicit Conversion

Implicit conversion happens automatically without explicit casting. Use for **safe**, **non-lossy** conversions.

Example from code:

```
public static implicit operator int(complex a)
{
    return a.real;
}
```

Usage:

```
complex c1 = new complex(3, 4);
int x = c1;  // Implicit conversion to int (no cast needed)
Console.WriteLine(x);  // Output: 3
```

What happens:

- Complex (3 + 4i) → int (3)
- Imaginary part is **lost** (but considered acceptable for implicit)
- No explicit cast syntax required

Explicit Conversion

Explicit conversion requires explicit casting syntax. Use for **potentially lossy** or **unsafe** conversions.

Example from code:

```
public static explicit operator float(complex a)
{
```

```
    return (a.real + (0.1F * a.img));
}
```

Usage:

```
complex c1 = new complex(3, 4);

// ❌ This won't compile without cast
// float x = c1;  // Compile error!

// ✅ Requires explicit cast
float x = (float)c1;  // x = 3 + (0.1 * 4) = 3.4
Console.WriteLine(x);  // Output: 3.4
```

When to Use Implicit vs Explicit:

Conversion Type	Use When	Example
Implicit	Safe, no data loss	int → long, float → double
Implicit	Obvious, natural	string → char[] (conceptually)
Explicit	Potential data loss	double → int, long → int
Explicit	Not obvious	Complex → float

More Conversion Examples:

```
// Implicit: int to complex (safe, natural)
public static implicit operator complex(int value)
{
    return new complex(value, 0);
}

// Usage:
complex c = 5;  // Automatically converts 5 to (5 + 0i)

// Explicit: complex to string representation
public static explicit operator string(complex c)
{
    return c.ToString();
}

// Usage:
string s = (string)c1;
```

```
// Implicit: bool conversion (for if statements)
public static implicit operator bool(complex c)
{
    return c.real != 0 || c.img != 0;  // True if non-zero
}

// Usage:
complex c = new complex(3, 4);
if (c)  // Uses implicit bool conversion
{
    Console.WriteLine("Complex is non-zero");
}
```

Conversion Best Practices:

1. **Prefer explicit over implicit** when in doubt
2. **Implicit should be obvious** and safe
3. **Document behavior** clearly (especially data loss)
4. **Consider bidirectional conversions**:

```
// int to complexpublic static implicit operator complex(int value) { }//
complex to intpublic static implicit operator int(complex c) { }
```

---

7. Indexers

Indexers allow objects to be indexed like arrays, providing intuitive access to internal collections.

What Are Indexers?

Indexers enable syntax like:

```
student s = new student(...);
s["C#"] = 40;           // Set value
int duration = s["C#"]; // Get value
```

Instead of:

```
s.SetSubjectDuration("C#", 40);
int duration = s.GetSubjectDuration("C#");
```

Basic Indexer Syntax:

```
public ReturnType this[ParameterType index]
{
    get
    {
        // Return value based on index
    }
    set
    {
        // Set value based on index and 'value' keyword
    }
}
```

String Indexer Example from Code:

```
public int this[string subjName]
{
    set
    {
        for (int i = 0; i < mysubjects.Length; i++)
        {
            if (mysubjects[i].name == subjName)
                mysubjects[i].duration = value;
        }
    }
    get
    {
        for (int i = 0; i < mysubjects.Length; i++)
        {
            if (mysubjects[i].name == subjName)
                return mysubjects[i].duration;
        }
        return 0;
    }
}
```

Usage:

```
subject[] mysub = new subject[]
{
    new subject(1, "C#", 60),
    new subject(2, "SQL", 40),
    new subject(3, "HTML", 6)
```

```
};

student s = new student(1, "ahmed mohamed", 20, mysub);

// Get duration
int duration = s["SQL"];  // Returns 40

// Set duration
s["C#"] = 80;  // Updates C# duration to 80

Console.WriteLine(s["SQL"]);  // Output: 40
```

How it works:

1. `s["SQL"]` calls the **getter** with `subjName = "SQL"`
2. Getter loops through `mysubjects` array
3. Finds subject with name "SQL"
4. Returns its duration (40)

Multiple Parameter Indexers:

Example from code:

```
public int this[int id, string subjName]
{
    set
    {
        for (int i = 0; i < mysubjects.Length; i++)
        {
            if (mysubjects[i].name == subjName)
                mysubjects[i].duration = value;
        }
    }
    get
    {
        for (int i = 0; i < mysubjects.Length; i++)
        {
            if (mysubjects[i].name == subjName)
                return mysubjects[i].duration;
        }
        return 0;
    }
}
```

Usage:

```
// Access with two parameters
s[1, "C#"] = 70;
int duration = s[1, "SQL"];
```

Indexer Overloading:

You can have multiple indexers with different parameter types:

```
class student
{
    // Indexer by string (subject name)
    public int this[string subjName] { get; set; }

    // Indexer by int (subject index)
    public subject this[int index]
    {
        get { return mysubjects[index]; }
        set { mysubjects[index] = value; }
    }

    // Indexer by two parameters
    public int this[int id, string subjName] { get; set; }
}

// Usage:
int duration1 = s["C#"];           // By name
subject subj = s[0];               // By index
int duration2 = s[1, "SQL"];       // By id and name
```

Read-Only Indexers:

```
public int this[string subjName]
{
    get
    {
        // Only getter, no setter
        for (int i = 0; i < mysubjects.Length; i++)
        {
            if (mysubjects[i].name == subjName)
                return mysubjects[i].duration;
        }
        return 0;
```

```
    }
    // No set accessor - read-only
}

// Usage:
int duration = s["C#"];  // ✅ Works
s["C#"] = 80;            // ❌ Compile error - no setter
```

Real-World Indexer Examples:

1. Dictionary-Like Access:

```
class Configuration
{
    private Dictionary<string, string> settings = new Dictionary<string,
string>();

    public string this[string key]
    {
        get { return settings.ContainsKey(key) ? settings[key] : null; }
        set { settings[key] = value; }
    }
}

// Usage:
Configuration config = new Configuration();
config["Theme"] = "Dark";
string theme = config["Theme"];
```

2. Matrix Access:

```
class Matrix
{
    private int[,] data;

    public int this[int row, int col]
    {
        get { return data[row, col]; }
        set { data[row, col] = value; }
    }
}

// Usage:
Matrix m = new Matrix(3, 3);
```

```
m[0, 0] = 1;
m[1, 1] = 5;
int value = m[0, 0];
```

3. Range-Checked Array:

```
class SafeArray
{
    private int[] data;

    public int this[int index]
    {
        get
        {
            if (index < 0 || index >= data.Length)
                throw new IndexOutOfRangeException();
            return data[index];
        }
        set
        {
            if (index < 0 || index >= data.Length)
                throw new IndexOutOfRangeException();
            data[index] = value;
        }
    }
}
```

---

8. StringBuilder Class

StringBuilder is a mutable string class optimized for string manipulation operations.

String vs StringBuilder:

Problem with String (Immutable):

```
string txt = "";
for (int i = 0; i < 1000; i++)
{
    txt += "Hello";  // Creates 1000 new string objects!
}
```

What happens:

- String is **immutable** (cannot be changed)
- Each concatenation creates a **new string object**
- Old strings become garbage
- **Very inefficient** for repeated modifications

Memory visualization:

```
Iteration 1: txt = "Hello"           [Object 1]
Iteration 2: txt = "HelloHello"      [Object 2, Object 1 → Garbage]
Iteration 3: txt = "HelloHelloHello" [Object 3, Object 2 → Garbage]
...
Result: 999 garbage objects!
```

Solution: StringBuilder (Mutable):

```
StringBuilder txt = new StringBuilder();
for (int i = 0; i < 1000; i++)
{
    txt.Append("Hello");  // Modifies same object!
}
string result = txt.ToString();
```

What happens:

- StringBuilder uses a **resizable buffer**
- Modifications happen **in-place** (no new objects)
- **Much more efficient** for repeated operations
- Convert to string only when needed

StringBuilder from Code:

Example:

```
public override string ToString()
{
    StringBuilder txt = new StringBuilder($"{id}-{name}-{age} years old
\nSubject:\n");

    for (int i = 0; i < mysubjects.Length; i++)
    {
        txt.AppendLine(mysubjects[i].ToString());
    }
```

```
    return txt.ToString();
}
```

Why StringBuilder here?

- Building string in a loop
- Multiple append operations
- More efficient than string concatenation

StringBuilder Methods:

1. **Append()** - Add to end:

```
StringBuilder sb = new StringBuilder();
sb.Append("Hello");
sb.Append(" ");
sb.Append("World");
// Result: "Hello World"
```

2. **AppendLine()** - Add with newline:

```
sb.AppendLine("Line 1");
sb.AppendLine("Line 2");
// Result: "Line 1\nLine 2\n"
```

3. **Insert()** - Insert at position:

```
StringBuilder sb = new StringBuilder("Hello World");
sb.Insert(6, "Beautiful ");
// Result: "Hello Beautiful World"
```

4. **Remove()** - Remove characters:

```
StringBuilder sb = new StringBuilder("Hello World");
sb.Remove(5, 6);  // Remove " World"
// Result: "Hello"
```

5. **Replace()** - Replace text:

```
StringBuilder sb = new StringBuilder("Hello World");
sb.Replace("World", "C#");
```

```
// Result: "Hello C#"
```

6. **Clear()** - Empty the StringBuilder:

```
sb.Clear();
// Result: ""
```

7. **ToString()** - Convert to string:

```
string result = sb.ToString();
```

StringBuilder Properties:

Capacity vs Length:

Example from code:

```
StringBuilder txt = new StringBuilder();
txt.Append("Hello World");

Console.WriteLine(txt.Capacity);  // e.g., 16 (buffer size)
Console.WriteLine(txt.Length);    // 11 (actual content)
```

- **Length**: Number of characters currently stored
- **Capacity**: Size of internal buffer (allocated memory)
- Capacity **grows automatically** when needed (usually doubles)

Capacity growth:

```
Initial capacity: 16
Add "Hello World" (11 chars): Capacity = 16, Length = 11
Add more text exceeding 16: Capacity doubles to 32
Add more text exceeding 32: Capacity doubles to 64
...
```

StringBuilder Constructors:

```
// Default (capacity 16)
StringBuilder sb1 = new StringBuilder();

// With initial string
```

```
StringBuilder sb2 = new StringBuilder("Hello");

// With initial capacity
StringBuilder sb3 = new StringBuilder(100);

// With initial string and capacity
StringBuilder sb4 = new StringBuilder("Hello", 100);

// With initial string, start index, length, and capacity
StringBuilder sb5 = new StringBuilder("Hello World", 0, 5, 50);
// Result: "Hello" with capacity 50
```

Performance Comparison:

```
// ❌ Bad: String concatenation (slow for many operations)
string result = "";
for (int i = 0; i < 10000; i++)
{
    result += i.ToString();  // Creates 10,000 objects!
}

// ✅ Good: StringBuilder (fast)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.Append(i);  // Modifies same object
}
string result = sb.ToString();
```

Performance difference:

- String concatenation: **O(n²)** time complexity
- StringBuilder: **O(n)** time complexity
- For 10,000 operations: StringBuilder is **hundreds of times faster**

When to Use StringBuilder:

✅ **Use StringBuilder when:**

- Building strings in loops
- Many concatenation operations (5+)
- String manipulation (insert, remove, replace)
- Performance is critical
- Working with large strings

✅ **Use String when:**

- Few concatenation operations (< 5)
- Simple, one-time concatenations
- String interpolation is sufficient
- Readability is more important than performance

Examples:

```
// ✅ String is fine (few operations)
string name = firstName + " " + lastName;
string greeting = $"Hello, {name}!";

// ✅ StringBuilder is better (many operations)
StringBuilder html = new StringBuilder();
for (int i = 0; i < items.Count; i++)
{
    html.Append("<li>");
    html.Append(items[i]);
    html.Append("</li>");
}
```

---

9. XML Documentation Comments

The code includes **XML documentation comments** for IntelliSense support.

Example from code:

```
/// <summary>
/// constructor to construct student objects
/// </summary>
/// <param name="id">student id</param>
/// <param name="name">student fullname</param>
/// <param name="age">student age</param>
/// <param name="mysubj">all student subjects</param>
public student(int id=0, string name=" ", int age=6, subject[] mysubj=null)
{
    // ...
}
```

XML Documentation Tags:

```
/// <summary>
/// Brief description of the member
/// </summary>
/// <param name="paramName">Description of parameter</param>
/// <returns>Description of return value</returns>
/// <exception cref="ExceptionType">When this exception is thrown</exception>
/// <remarks>Additional remarks or notes</remarks>
/// <example>
/// Example usage:
/// <code>
/// var result = MyMethod(5);
/// </code>
/// </example>
public int MyMethod(int paramName)
{
    return paramName * 2;
}
```

Benefits:

1. **IntelliSense tooltips** in Visual Studio
2. **API documentation** generation
3. **Better code understanding** for other developers
4. **Compiler warnings** for missing documentation

Key Takeaways Summary

1. **Operator Overloading**: Define custom behavior for operators (+, -, *, ==, etc.) on your types
2. **Binary Operators**: Work with two operands (a + b), must be `public static`
3. **Unary Operators**: Work with one operand (++a, -a)
4. **Comparison Operators**: Must overload in pairs (== with !=, < with >)
5. **Non-Overloadable**: Cannot overload =, +=, [], ., &&, ||, ?:, etc.
6. **Implicit Conversion**: Automatic, safe type conversion (no cast needed)
7. **Explicit Conversion**: Requires cast syntax, for potentially lossy conversions
8. **Indexers**: Enable array-like syntax on objects (obj[key])
9. **Multiple Indexers**: Can overload with different parameter types
10. **StringBuilder**: Mutable string class for efficient string building
11. **Capacity vs Length**: StringBuilder buffer size vs actual content

12. **Performance**: StringBuilder is O(n) vs String concatenation O(n²)

These concepts enable creating **intuitive**, **performant**, and **elegant** APIs for your custom types!

By Abdullah Ali

Contact : +201012613453