## **Operator Overloading**

**Operator overloading** allows you to redefine or provide a custom implementation for an operator when it's used with user-defined classes or structs. The primary goal is to make operations on custom types as natural and readable as operations on built-in types (like int or string).

Operator overloading is most appropriately used for **structs that represent primitive-like data types**. A classic example is a custom numeric type, a complex number, or, as we'll see, a musical note. It should enhance readability, not obscure meaning.

### **Overloadable Operators**

Most common symbolic operators can be overloaded:

* **Unary Operators:** + (unary), - (unary), !, ~, ++, --
* **Binary Operators:** +, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

Additionally, the following special operators can be overloaded:

* **Implicit and Explicit Conversions:** Using the implicit and explicit keywords.
* **true and false Operators:** These are not the boolean literals, but special operators for types that behave like booleans.

**Operators that are indirectly overloaded:**

* **Compound Assignment Operators:** (+=, -=, \*=, /=, etc.) are automatically overloaded if you overload their corresponding non-compound operators (+, -, \*, /).
* **Conditional Operators:** (&&, ||) are implicitly overridden if you overload the bitwise operators (&, |) and the true/false operators.

### **Operator Functions**

To overload an operator, you declare a special method called an **operator function**. These functions must adhere to specific rules:

* **Syntax:** The function name is specified using the operator keyword followed by the operator symbol (e.g., operator +).
* **Modifiers:** They must be marked static and public.
* **Parameters:** The parameters represent the operands of the operation.
* **Return Type:** The return type represents the result of the operation.
* **Operand Requirement:** At least one of the operands *must* be of the type in which the operator function is declared. This prevents you from overloading operators for built-in types (e.g., you can't overload int + int).

**Example: Overloading the + Operator for a Note struct**

Let's define a Note struct to represent a musical note, where value represents semitones from A.

| public struct Note {  int value; // Semitones from a base 'A' note   public Note(int semitonesFromA) { value = semitonesFromA; }   // Overloading the '+' operator  // This allows us to add an integer (semitones) to a Note  public static Note operator +(Note x, int semitones)  {  return new Note(x.value + semitones);  } } |
| --- |

Now, you can add an int to a Note instance just like you would with numbers:

| Note B = new Note(2); // B is 2 semitones from A Note CSharp = B + 2; // CSharp is now a Note representing (2 + 2) = 4 semitones from A |
| --- |

Because we overloaded +, the corresponding compound assignment operator += is automatically available:

| CSharp += 2; // Equivalent to CSharp = CSharp + 2 |
| --- |

Like regular methods, operator functions can also be written using **expression-bodied syntax** for conciseness:

| public static Note operator +(Note x, int semitones) => new Note(x.value + semitones); |
| --- |

### **Checked Operators (C# 11+)**

From C# 11, you can define a checked version of an operator function alongside its unchecked counterpart. This allows you to provide overflow-checking behavior explicitly for your overloaded operators.

| public static Note operator +(Note x, int semitones)  => new Note(x.value + semitones); // Unchecked version  public static Note operator checked +(Note x, int semitones)  => checked(new Note(x.value + semitones)); // Checked version |
| --- |

The checked version will be invoked when the operation is performed within a checked expression or block:

| Note B = new Note(2); Note other = checked(B + int.MaxValue); // This will call the 'checked +' operator  // and likely throw an OverflowException |
| --- |

### **Overloading Equality and Comparison Operators**

Overloading ==, !=, <, >, <=, and >= requires special consideration and adherence to certain conventions to ensure consistent and correct behavior.

**Summary of Rules:**

* **Pairing:** The C# compiler enforces that **logical pairs of operators must both be defined**.
  + If you overload ==, you *must* also overload !=.
  + If you overload <, you *must* also overload >.
  + If you overload <=, you *must* also overload >=.
* **Equals and GetHashCode:** In most cases, if you overload == and !=, you **must also override the Equals(object obj) and GetHashCode() methods** inherited from System.Object. The C# compiler will issue a warning if you don't. This ensures that equality checks via operators, Equals method calls, and hash-based collections (like Dictionary and HashSet) all behave consistently.
* **IComparable and IComparable<T>:** If you overload the comparison operators (<, >, <=, >=), you **should implement the System.IComparable and System.IComparable<T> interfaces**. This provides a standard way for your type to be compared using methods like CompareTo(), which is useful for sorting.

## **Custom Implicit and Explicit Conversions**

You can also overload the implicit and explicit conversion operators. These are useful for making conversions between strongly related types natural and concise.

* **implicit conversions:** Should be used only when the conversion is guaranteed to succeed and no information will be lost. The compiler performs these automatically.
* **explicit conversions:** Should be required when the conversion might fail at runtime or might result in a loss of information. The user must explicitly cast.

**Example: Converting between Note and double (representing frequency)**

| public struct Note {  int value; // Semitones from A (e.g., A=0, A#=1, B=2...)  public Note(int semitonesFromA) { value = semitonesFromA; }   // Implicit conversion from Note to double (frequency in Hertz)  // No information loss, always succeeds.  public static implicit operator double(Note x)  => 440 \* Math.Pow(2, (double)x.value / 12);   // Explicit conversion from double (frequency) to Note  // Information might be lost (rounding to nearest semitone), so explicit is required.  public static explicit operator Note(double x)  => new Note((int)(0.5 + 12 \* (Math.Log(x / 440) / Math.Log(2)))); } |
| --- |

Usage:

| Note n = (Note)554.37; // Explicit conversion from double to Note double x = n; // Implicit conversion from Note to double |
| --- |

**Important:** Custom conversions are **ignored** by the as and is operators. You cannot use is Note or as Note with a value of a different type that only has custom conversions defined.

## **Overloading true and false Operators**

Overloading the true and false operators is an extremely rare scenario, typically reserved for types that conceptually represent a Boolean value but don't have a direct conversion to bool. This enables such types to work seamlessly with conditional statements (if, while) and logical operators (&&, ||).

A prime example is System.Data.SqlTypes.SqlBoolean, which implements three-valued logic (True, False, Null).

| using System.Data.SqlTypes;  SqlBoolean a = SqlBoolean.Null;  if (a) // Uses the overloaded operator true  Console.WriteLine("True"); else if (!a) // Uses the overloaded operator false and operator !  Console.WriteLine("False"); else  Console.WriteLine("Null"); // Output: Null |
| --- |

By overloading operator true and operator false, SqlBoolean can participate in standard conditional constructs, even though its internal representation isn't a simple bool.

| // Reimplements parts of SqlBoolean for demonstration public struct SqlBoolean {  public static bool operator true(SqlBoolean x)  => x.m\_value == True.m\_value; // Returns true if the internal value matches 'True'   public static bool operator false(SqlBoolean x)  => x.m\_value == False.m\_value; // Returns true if the internal value matches 'False'   public static SqlBoolean operator !(SqlBoolean x)  {  if (x.m\_value == Null.m\_value) return Null;  if (x.m\_value == False.m\_value) return True;  return False;  }   public static readonly SqlBoolean Null = new SqlBoolean(0);  public static readonly SqlBoolean False = new SqlBoolean(1);  public static readonly SqlBoolean True = new SqlBoolean(2);   private SqlBoolean(byte value) { m\_value = value; }  private byte m\_value; } |
| --- |