

C# LANGUAGE SPECIFICATION

The `bool` type is used to represent Boolean values: values that are either true or false. The inclusion of `bool` makes it easier to write self-documenting code; it also helps eliminate the all-too-common C++ coding error in which a developer mistakenly uses “=” when “==” should have been used. In C#, the example

```
int i = ...;
F(i);
if (i = 0) // Bug: the test should be (i == 0)
    G();
```

results in a compile-time error because the expression `i = 0` is of type `int`, and `if` statements require an expression of type `bool`.

The `char` type is used to represent Unicode code units. A variable of type `char` represents a single 16-bit Unicode code unit.

The `decimal` type is appropriate for calculations in which rounding errors caused by floating point representations are unacceptable. Common examples include financial calculations such as tax computations and currency conversions. The `decimal` type provides for at least 28 significant digits.

The table below lists the predefined types, and shows how to write literal values for each of them.

Type	Description	Example
<code>object</code>	The ultimate base type of all other types	<code>object o = null;</code>
<code>string</code>	String type; a string is a sequence of Unicode code units	<code>string s = "hello";</code>
<code>sbyte</code>	8-bit signed integral type	<code>sbyte val = 12;</code>
<code>short</code>	16-bit signed integral type	<code>short val = 12;</code>
<code>int</code>	32-bit signed integral type	<code>int val = 12;</code>
<code>long</code>	64-bit signed integral type	<code>long val1 = 12;</code> <code>long val2 = 34L;</code>
<code>byte</code>	8-bit unsigned integral type	<code>byte val1 = 12;</code>
<code>ushort</code>	16-bit unsigned integral type	<code>ushort val1 = 12;</code>
<code>uint</code>	32-bit unsigned integral type	<code>uint val1 = 12;</code> <code>uint val2 = 34U;</code>
<code>ulong</code>	64-bit unsigned integral type	<code>ulong val1 = 12;</code> <code>ulong val2 = 34U;</code> <code>ulong val3 = 56L;</code> <code>ulong val4 = 78UL;</code>
<code>float</code>	Single-precision floating point type	<code>float val = 1.23F;</code>
<code>double</code>	Double-precision floating point type	<code>double val1 = 1.23;</code> <code>double val2 = 4.56D;</code>
<code>bool</code>	Boolean type; a <code>bool</code> value is either true or false	<code>bool val1 = true;</code> <code>bool val2 = false;</code>
<code>char</code>	Character type; a <code>char</code> value is a Unicode code unit	<code>char val = 'h';</code>
<code>decimal</code>	Precise decimal type with at least 28 significant digits	<code>decimal val = 1.23M;</code>

Each of the predefined types is shorthand for a system-provided type. For example, the keyword `int` refers to the struct `System.Int32`. As a matter of style, use of the keyword is favored over use of the complete system type name.

Predefined value types such as `int` are treated specially in a few ways but are for the most part treated exactly like other structs. Operator overloading enables developers to define new struct types that behave much like the predefined value types. For instance, a `Digit` struct can support the same mathematical