Simple Value Types

Simple value types have special representation and are directly supported by the virtual execution system. Many are frequently used, and the chart in Table 4.1 will help determine the size of each.

Category	Bits	Type	Description
Integral types	32*	System.Boolean	True/False
	16	System.Char	Unicode 16-bit character
	8	System.SByte	Signed 8-bit integer
	16	System.Int16	Signed 16-bit integer
	32	System.Int32	Signed 32-bit integer
	64	System.Int64	Signed 64-bit integer
	32/64	System.IntPtr	Signed native integer
	8	System.Byte	Unsigned 8-bit integer
	16	System.UInt16	Unsigned 16-bit integer
	32	System.UInt32	Unsigned 32-bit integer
	64	System.UInt64	Unsigned 64-bit integer
	32/64	System.UIntPtr	Unsigned native integer
Floating point types	32	System.Single	32-bit float
	64	System.Double	64-bit float
Typed reference	64/128	System.TypedReference	Pointer and exact type

Table 4.1: Comparison of various value types.

System.Boolean

Although `System.Boolean` is an integral type at the CLI level, C# provides no conversion to and from `int`. C++ does provide this functionality: `false` is treated as 0 and `true` is treated as 1. This is important from an interoperability perspective.

This is significant because, while `System.Boolean` is 1 byte (8 bits) in managed code, when marshaled to Windows it will be converted to the native Windows BOOL format, which is 4 bytes. Marshaling refers to converting blocks of memory from managed to unmanaged code.

```
Assert.AreEqual(1, sizeof(Boolean));
Assert.AreEqual(4, Marshal.SizeOf(new Boolean()));
```

Listing 4.14: Some surprises on the size of a Boolean.

The size of a Boolean may seem a little surprising. Since it has two possible values, you would expect it to be a single bit. The extra size has more to do with performance than with memory efficiency, as the extra space is used to resolve alignment issues and speed up data access. If you have a struct or object with lots of `Boolean` fields, you may want to consider a `BitVector32`, but this will introduce more complex code. You need to make sure that the memory gains justify this extra complexity.

System.Char

`System.Char` is 2 bytes but `Marshal.SizeOf` reports it as 1 byte. `System.Char` is Unicode, whereas a Windows char is your standard ASCII character. This is important if you are supporting non-English characters. This size difference can create interoperability problems and may throw off your estimates for how much memory will be used if you expect each character to be a single byte.

```
Assert.AreEqual(2, sizeof(Char));
Assert.AreEqual(1, Marshal.SizeOf(new Char()));
```

Listing 4.15: Comparing the sizes of chars in managed and unmanaged code.

Overflow checking

The .NET framework can do automatic arithmetic overflow checking to ensure that calculations are not corrupted by overflowing the values in the underlying type system. By default this is disabled in C#. Ironically, this is enabled by default for a VB.NET project, but there is little risk from disabling it in most cases. As you will see below, the right set of conditions have to be met to even have a risk of an overflow.

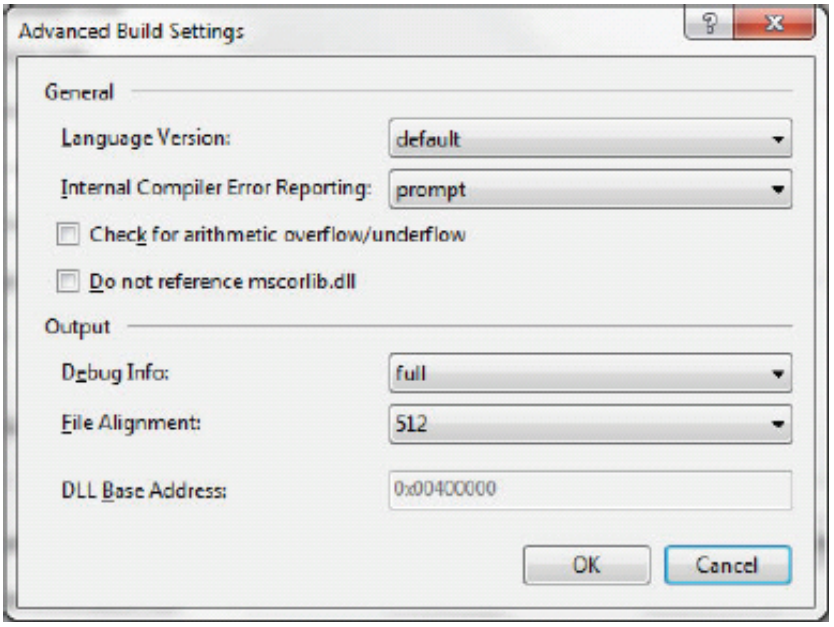

Figure 4.2: You can explicitly enable/disable overflow checking.

The maximum value of `System.Int32` is 2,147,483,647. What is the value when you add `Int32.MaxValue` to itself? Well, the compiler is smart enough to throw a compile-time error: *The operation overflows at compile time in checked mode.* Of course, the compiler can easily be confused. Let's add a method (see Listing 4.16).

```
public int ForceAnError()
{
 return Int32.MaxValue + Int32.MaxValue;
```

```
}
public int Add(int x, int y)
{
 return x + y;
}
```

Listing 4.16: Forcing an overflow exception.

The `ForceAnError` method will generate a compile-time error even though the **Check for overflow/underflow** option is not checked. This is a runtime setting and this error can be detected at compile time. This advanced build setting is intended only to avoid the overhead of an unnecessary check.

Now we call the `add` method, passing in `Int32.MaxValue` for both parameters. We don't get a compile-time error because the compiler is simply not that smart, but we also don't get the expected result. We get an overflow calculation and the wrong answer. What answer do we get? Amazingly, we get -2.

Overflowing `Int32.MaxValue` by 1 will make the variable `Int32.MinValue`, -2,147,483,648. That leaves 2,147,483,646 left in the second variable. Add them together to get -2. The reason why `Int32.MaxValue + 1` results in `Int32.MinValue` is because of "Two's Complement" arithmetic. This is how negative numbers are represented in a binary system. In a Two's Complement representation, the Most Significant Bit (MSB) in the number is always 1. A positive number has the normal binary representation for the number with leading zeros to fill up to the MSB. To convert this to a negative number, the bits are inverted with a bitwise `NOT` operation. We then add 1 to the resulting value. In binary, `Int32.MaxValue = 0111 1111 1111 1111 1111 1111 1111 1111.` Following this formula, `Int32.MinValue = 1000 0000 0000 0000 0000 0000 0000 0000.` Looking at the two values in binary, you can hopefully see how adding 1 to `Int32.MaxValue` results in `Int32.MinValue`.

```
Assert.AreEqual(-2, Add(Int32.MaxValue, Int32.MaxValue));
```

Listing 4.17: The surprising result of adding MaxInt to itself.

Such overflows are allowed for performance reasons, but they can lead to logical bugs that would be difficult to track down. You have the option to enable the check for overflow/ underflow at the assembly level, and then every calculation that takes place in that assembly will be checked. Alternatively, when performance is preferred but some operations should be checked, use the `checked` keyword to check specific calculations without incurring this extra cost everywhere.

```
public int Add(int x, int y)
{
 return checked(x + y);
}
```

Listing 4.18: Explicitly checking a key calculation for overflow.

An `OverflowException` is now thrown at runtime when the above method is called passing in `MaxValue` for the parameters. If you really want to allow for overflowing, such as adding `Int32.MaxValue` to itself, there's a corresponding `unchecked` keyword.