CHAPTER 11

Expressions and Operators

Argue for your limitations, and sure enough, they're yours.

—Richard Bach, *Illusions*

In this chapter, we'll cover expressions and operators in the context of how they differ from C#. We'll start with a caveat: don't assume that expression evaluation is the same in C# and C++. C# and C++/CLI have different rules that control the evaluation of expressions, and this can surprise you if you're writing overly complex expressions.

Here is an old C++ trick that often shows up on interview questions. It is not guaranteed to work according to the C++ standard, but it works on all of the major C++ compilers for the x86. It is called the XOR swap, and it allows you to swap the values of two integers without declaring an explicit temporary. The code follows:

```
using namespace System;
void main()
{
    int i=3, j=6;
    Console::WriteLine("{0}, {1}", i, j);
    i ^= j ^= i ^= j;
    Console::WriteLine("{0}, {1}", i, j);
}
```

Let's run it:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
3, 6
6, 3
```

Look at the following line:

```
i ^= j ^= i ^= j;
```

As you can see, it swaps the values of `i` and `j`, because it evaluates the same as the following:

```
i ^= j;
j ^= i;
i ^= j;
```

The first XOR swap stores the bitwise difference between `i` and `j` in `i`. The next line changes `j` by this difference, turning it into `i`. The last line, in turn, changes what's left of `i` into `j` by changing what was originally `i` (currently `j`) by the difference (currently `i`) too. This relies on the following identities for exclusive OR:

```
x == y ^ (x^y)
```

It more or less breaks up `x` and `y` into two parts: the part they have in common, and the part that differs, like taking two numbers and knowing they are equidistant from their average.

Now let's try to do this in C#:

```
using System;
class R
{
    public static void Main()
    {
        int i=3, j=6;
        Console.WriteLine("{0}, {1}", i, j);
        i ^= j ^= i ^= j;
        Console.WriteLine("{0}, {1}", i, j);
    }
}
```

The results follow:

```
C:\>csc /nologo test.cs
C:\>test
3, 6
0, 3
```

As you can see, this just does not work. It's probably just a lack of parentheses, right? What if we try this?

```
i ^= (j ^= (i ^= j));
```

That won't work either; we get the same results. The answer here is that C# and C++ evaluate expressions differently. The rules are pretty complicated, and you don't really need to know them that well unless your chosen vocation is rules lawyer.

■ **Note** In this case, the C# code evaluates differently, because C# separates the evaluation of an expression with the evaluation of a variable, in order to help optimizers. C++ evaluates expressions as parenthesized; C# is free to scan the entire statement, preevaluate variables, and use those values to evaluate the expression. This code relies on the interim values of `i` and `j` being updated in the middle of the expression in order to work correctly.

It's good programming practice simply to avoid these esoteric constructs. The easy and safe way to write code that works correctly with both languages is to subdivide the expression:

```
i ^= j;
j ^= i;
i ^= j;
```

This sequence works correctly in both C# and C++. Ten years ago, weaving these expressions together might have produced faster code; today's optimizing compilers are sophisticated enough to figure out what you're trying to do and compensate for the expansion.

Operator Overloading

One of the most important aspects of C# and C++/CLI is the support that they offer for elevating user-defined types to the level of built-in types; one important aspect of this is the ability to define operators to work with new types. The examples most commonly encountered in the published literature define types for complex variables or fractions, but that is really just the tip of the iceberg. It is also common practice to define operators to perform operations that have absolutely nothing to do with their mathematical definitions, pushing the boundaries of our limited paradigms and often redefining new ones.

Of course, there are limitations, which include the following:

- Unary operators must remain unary; binary operators must remain binary. In other words, you cannot redefine the plus sign (+) to take three arguments instead of two.
- You cannot make up operators that do not exist. You cannot define a `/%` operator, even though it could logically be disambiguated by the grammar. You are limited to the language's built-in operators.
- You cannot control the predefined order of evaluation, and you can't expect complicated expressions to evaluate the same way across C++/CLI and C#. As mentioned before, C# and C++/CLI have different rules that control the evaluation of expressions.

Complex Numbers, a Basic Example

Recall that we can consider a simple complex number class in C++/CLI and that complex numbers are numbers of the form:

$$a + bi$$

where

$$i = \sqrt{-1}$$

This helps us to lay a foundation for Chapter 15, when we revisit complex numbers using templates in the context of numbers of the form:

$$(a + b\sqrt{5})$$

This form is very useful when working with the golden ratio:

$$\phi = \frac{1}{2}(1 + \sqrt{5})$$

Using the golden ratio, we can calculate the Fibonacci numbers with a nonrecursive, simple, closed form.[1]

[1]Knuth, Donald E. *Art of Computer Programming, vol. 1: Fundamental Algorithms*, 3rd ed. (Boston: Addison-Wesley, 1997).

A Review of Complex Numbers

A review of the fundamental mathematical operations using complex numbers follows:[2]

Addition:

$$(a + bi) + (c + di) = (a + c) + (b+d)i$$

Subtraction:

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

Complex conjugation:

$$\overline{(a+bi)} = (a - bi)$$

Multiplication:

$$(a + bi) \times (c + di) = (ac - bd)+(ad + bc)i$$

Division by a scalar (real) number:

$$\frac{(a+bi)}{m} = \frac{a}{m} + \frac{b}{m}i$$

Division between complex numbers: Using complex conjugation, multiplication, division by a scalar, and the following identity,

$$(c+di)(\overline{c+di}) = (c^2 + d^2)$$

we can derive division between complex numbers:

$$\frac{(a+bi)}{(c+di)} = \frac{(a+bi)\times(\overline{c+di})}{(c+di)\times(\overline{c+di})}$$

Note how this divide operation defers to complex conjugation as well as multiplication for calculation of the quotient.

Simple Implementation

We implement this class by defining class data as well as the operators that act on the data. The class data is, very simply, two doubles that correspond to the real part and the imaginary part, which is multiplied by the following:

$$i = \sqrt{-1}$$

[2]These are derived from the associative, commutative, and distributive laws of the field of complex numbers, along with the definition of `i`.

The data structure follows:

```
value struct Complex
{
    double re;
    double im;
}
```

As for the operators themselves, there are several ways to define them, depending on whether we want our code to be Common Language Specification (CLS) compliant (we'll revisit this later in the chapter). Essentially, our operators are static member functions, and they return objects rather than references.

Unary Operators

CLI unary operations have the following format:

```
static type operator op (type a)
```

We'll use the following operator in our class:

Complex conjugation:

```
static Complex operator ~ (Complex a);
```

Binary Operators

CLI binary operations have the following format:

```
static type operator op (type a, type b)
```

We'll use these operators in our class:

Addition:

```
static Complex operator + (Complex a, Complex b);
```

Subtraction:

```
static Complex operator - (Complex a, Complex b);
```

Multiplication:

```
static Complex operator * (Complex a, Complex b);
```

Division by double:

```
static Complex operator / (Complex a, double b);
```

Division by complex:

```
static Complex operator / (Complex a, Complex b);
```

Order Matters

Note that the code makes no assumption of commutativity; it is perfectly reasonable to define `a/b` to be different from `b/a`, so it is also possible to implement the following line:

```
static Complex operator / (Complex a, double b)
```

in a different method from this line:

```
static Complex operator / (double a, Complex b)
```

The Product of Our Efforts

The completed program follows:

```
using namespace System;
value struct Complex
{
    double re;
    double im;
    Complex(double re, double im)
    {
        this->re = re;
        this->im = im;
    }
    static Complex operator + (Complex a, Complex b)
    {
        return Complex(a.re+b.re, a.im+b.im);
    }
    static Complex operator - (Complex a, Complex b)
    {
        return Complex(a.re-b.re, a.im-b.im);
    }
    static Complex operator ~ (Complex a)
    {
        return Complex(a.re, - a.im);
    }
    static Complex operator * (Complex a, Complex b)
    {
        return Complex(a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re);
    }
    static Complex operator / (Complex a, Complex b)
    {
        return a / (b.re*b.re+b.im*b.im) * ~b;
    }
    virtual String ^ ToString() override
    {
        String ^s = re.ToString();
        if(im != 0)
        {
            return s += " + " + im.ToString() + "i";
        }
```

```
        return s;
    }
private:
    static Complex operator / (Complex a, double f)
    {
        return Complex(a.re/f, a.im/f);
    }
};
void main()
{
    Complex a(-5,10), b(3,4);
    Console::WriteLine("({0}) / ({1}) = {2}",a,b,a/b);
}
```

As you can see, the basic operators +, -, *, and / have been overloaded to operate on `Complex` types rather than the subtypes they are based on, which in this case is `double`.

The unary complement operator ~ from Boolean logic doesn't intuitively correspond to any operation that you would perform on a complex number over the real numbers. It is, therefore, an ideal candidate for satisfying our need for a unary operator for complex conjugation, which we need to implement `operator/`. The compiler does not enforce any logical paradigm beyond number of arguments and argument type. You are free to define `operator*` as division and `operator/` as multiplication. This, of course, is bad form unless obfuscation is your goal.

The results from a quick compile and run follow:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
(-5 + 10i) / (3 + 4i) = 1 + 2i
```

Resolution of Overloads

You might also have noticed that there are two different `operator/` methods for division. Both C# and C++ have built-in rules for choosing which method to call, but these differ in surprising ways. Two methods that share the same name yet have different parameters are called *overloads*. The process of determining the closest match for a given set of parameters is called *overload resolution*, and I introduce it here in the context of `operator/`, though it will continue to be a topic we cover in passing.

Suppose we replace the previous `main()` function with the following one:

```
void main()
{
    Complex a(-5,10);
    float b = 5.0f;
    Console::WriteLine("({0}) / ({1}) = {2}",a,b,a/b);
}
```

Let's run this example:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
(-5 + 10i) / (5) = -1 + 2i
```

When this example is executed, the compiler needs to look up how to calculate `a/b`, where the variable `a` is of type `Complex`, and the variable `b` is of type `float`.

The compiler parses `a/b` and begins to look for a compatible method of the form

```
operator/(Complex a,float b)
```

In the source of the program, there is no method that has this exact signature, so the compiler gathers up a list of possible candidate methods and tries to determine the best match. In this case, the possible choices are as follows:

```
operator/(Complex a,double b)
operator/(Complex a,Complex b)
```

Neither of these is an exact match. There are well-defined rules within the C++ standard governing the resolution of overloads, and these rules apply not only to operators but to functions in general. I don't want to dwell too much on this; for now, know that the intuitive choice in this case is the winner. A permitted operation is to promote (extend) the `float` to a `double` and choose the following:

```
operator/(Complex a,double b)
```

The rules for overload resolution provide a multitiered approach to dealing with implicit and explicit conversions. Certain conversions are favored over other conversions, and this is anything but arbitrary. At first glance, it does not seem to be a topic fraught with danger, but consider the following: Suppose we added an implicit conversion from `double` to `Complex` to our code. If the compiler could perform this conversion automatically, would we have to worry about creating an infinite recursion? Since `operator/(Complex, Complex)` calls `operator/ (Complex, double)`, adding an implicit conversion from `double` to `Complex` might result in either an ambiguity or an infinite loop. In this case, it does not because of the prioritization rules in the C++ specification, which assign a rank to each type of conversion and prioritize them by rank. We'll discuss implicit and explicit conversions later on in the chapter.

Just when you thought that the subject of complex numbers was getting too mathematical, brace yourself—I am pleased to present the following mathematical diversion.

A Mathematical Diversion: Numbers Modulo Primes

Both C# and C++/CLI use the percent sign as an operator for the calculation of one number modulo a different number. Recall that `(number%p)` is equal to the remainder when `number` is divided by `p`. It is easy to define a class of numbers modulo a number `p`. The following is simply the set of numbers:

$$\{0,1,\ldots(p-1)\}$$

Now we just need to figure out how to perform operations on them.

We can easily redefine the basic operators of addition, multiplication, and subtraction by calculating the result modulo `p`. Division would normally cause us to run into the use of fractions, but a result from elementary number theory tells us that division can be defined without fractions when the modulus `p` is prime. For example, let's consider the numbers modulo 13, and suppose we're trying to figure out what a quarter, 1 divided by 4, is. In other words, what is the inverse of 4?

A simple calculation shows that `(4 * 10) % 13 = 1`, since `4*10=40=39+1`, thus 1 is the remainder when 40 is divided by 13.

Let's use the compiler to prove this:

```
using namespace System;
void main()
{
    Console::WriteLine("4 * 10 = {0} (13)", 4*10%13);
}
```

When we compile and execute this, we have the following:

```
C:\>cl /nologo /clr:pure test.cpp
C:\>test
4 * 10 = 1 (13)
```

10 is the inverse of 4. If we divide both sides by 4, we get the following:

$$\frac{1}{4} \equiv 10 \text{ modulo } 13$$

because

$$4 \times 10 = 1 + 13 \times 3 \equiv 1$$

Likewise, all nonzero numbers modulo 13 have an inverse in the same way. To find it, we have to use another result from number theory. It turns out, that for every two numbers a and b, there exist numbers x and y such that

$$ax + by = \gcd(a, b)$$

If one of the numbers is a prime, and the other is not a multiple of this prime, then the greatest common divisor (gcd) of these two numbers is 1, and we have the following:

$$ax + py = 1$$

Since any multiple of p is 0 by definition, we get

$$py \equiv 0 \text{ modulo } p$$

Read the preceding expression as follows: py is congruent to 0 modulo p, since py has remainder 0 when divided by p. Combining these facts, we conclude that there exists a number x such that

$$ax \equiv 1 \text{ modulo } p$$

In other words, we just need to find the number *x*, and we have our inverse! I won't bore you with any more details or derivations, but there is an extended version of the Euclidean algorithm that will do this for you.[3] It's in the following code; note that `ExtendedEuclid()` is implemented as a global function, not a class method, and it takes references to integers as some of its parameters:

```
using namespace System;
void ExtendedEuclid(int a, int b, int %d, int %x, int %y)
{
    if(b==0)
    {
        d=a;
        x=1;
        y=0;
    }
    else
    {
        ExtendedEuclid(b,a%b, d, y, x);
        y-= (a/b)*x;
    }
}
value struct F13
{
    unsigned Value;
    initonly static unsigned P = 13;
    F13(unsigned Val)
    {
        Value = Val % P;
    }
    static F13 operator * (F13 arg1, F13 arg2)
    {
        return F13((arg1.Value * arg2.Value) % P);
    }
    static F13 operator + (F13 arg1, F13 arg2)
    {
        return F13((arg1.Value + arg2.Value) % P);
    }
    static F13 operator - (F13 arg1, F13 arg2)
    {
        return F13((arg1.Value - arg2.Value) % P);
    }
    static F13 operator - (F13 arg1)
    {
        return F13((P - arg1.Value) % P);
    }
```

[3]Cormen, Thomas H. *Introduction to Algorithms*, 2nd ed. (Cambridge, MA: MIT Press, 2001).

```
    static F13 operator / (F13 arg1, F13 arg2)
    {
        int d, x, y;
        ExtendedEuclid(arg2.Value,P,d,x,y);
        return arg1*F13(x*d);
    }
    virtual String ^ ToString() override
    {
        Value = (Value+P) % P;
        String ^s = Value.ToString();
        return s;
    }
};
void main()
{
    F13 a(6), b(9), c(4), d(10);
    Console::WriteLine("{0} * {1} is {2}", a, b, a*b);
    Console::WriteLine("{0} / {1} is {2}", a, b, a/b);
    Console::WriteLine("{0} * {1} is {2}", c, d, c*d);
}
```

And here are the results:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
6 * 9 is 2
6 / 9 is 5
4 * 10 is 1
```

Implicit and Explicit Conversions of Built-in Types

Both C# and C++/CLI support defining implicit and explicit conversions between types. This is the user-defined type equivalent of promoting a `float` to a `double` or a `short` to an `int`. Implicit conversions are the conversions that the compiler can apply automatically, where explicit conversions require a cast operator. Let's talk a little bit about conversions between built-in types.

Conversion Differences Between C# and C++

Unfortunately, the implicit conversions over the built-in types differ between C++ and C#. C++ is historically notoriously lax in preventing conversions that risk data loss. Consider the following example:

```
using namespace System;
void main()
{
    long l=65537;
    short s=0;
    s=l;
    l=s;
    Console::WriteLine(l);
}
```

Now let's give this a try:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
1
```

In this case, the compiler implicitly converted between `short` and `long`, and vice versa, without a warning as to the possible loss of data. If we raise the warning level to 3, we get the following output:

```
C:\>cl /clr:pure /nologo /W3 test.cpp
test.cpp(6) : warning C4244: '=' : conversion from 'long' to 'short', possible loss
of data
```

Now that's more like it!

Suppose we change the `long` to an `int` and compile at warning level 3. The data loss remains:

```
C:\>cl /clr:pure /nologo /W3 test.cpp
C:\>test
1
```

At best, that's an annoyance. At worst, it's a recall-class bug. Luckily, if we boost the warning level to level 4, we get the following:

```
C:\>cl /clr:pure /nologo /W4 test.cpp
test.cpp
test.cpp(6) : warning C4244: '=' : conversion from 'int' to 'short', possible loss
of data
```

The only problem with level-4 warnings is that they are considered more advisory than diagnostic, and sometimes come up as spurious or noisy warnings. The lesson learned is that you need to be careful. The C++ compiler isn't watching your back as much as the C# compiler in this area, and as I've recommended before, when you're developing your code, turn on `/W4` warnings every now and then.

Signed/Unsigned Mismatches

The C++/CLI compiler does have the ability to warn if you attempt to assign a signed value to an unsigned variable and vice versa. It is disabled by default, but can be enabled using the `/Wall` compiler option, which enables all warnings that are disabled by default in the compilation.

For example, consider the following:

```
void main()
{
    unsigned u=0;
    int i=0;
    i=u;
}
```

After compiling, we get

```
C:\>cl /Wall /nologo test.cpp
test.cpp(5) : warning C4365: '=' : conversion from 'unsigned int' to 'int',
signed/unsigned mismatch
```

Integer Conversion Tables

Let's go over some of the built-in conversions in C++ and C#. Use the following list of abbreviations as a key to interpret the integer conversion tables (Table 11-1 to Table 11-5):

ex: Explicit

im: Implicit, no warning

i2: Implicit, warning level 2

i3: Implicit, warning level 3

i4: Implicit, warning level 4

ia: Implicit, warning `/Wall` only (to indicate signed/unsigned mismatch)

X: No conversion needed

Let's first look at the integer types in Table 11-1.

Table 11-1. *C++/CLI Conversion Table for a Sampling of Built-in Integer Types*

C++	**From**					
To	**From** `short`	**From** `int`	**From** `long`	**From** `long long`	**From** `unsigned int`	**From** `unsigned long long`
To `short`	X	i4	i3	i3	i4	i3
To `int`	im	X	im	i3	ia	i3
To `long`	im	im	X	i3	ia	i3
To `long long`	im	im	im	X	im	ia
To `unsigned int`	ia	ia	ia	i3	X	i3
To `unsigned long long`	ia	ia	ia	ia	ia	X

Let's look at the C# table for the integer types in Table 11-2.

Table 11-2. *C# Conversion Table for a Sampling of Built-in Integer Types*

C++	From				
To	**From** `short`	**From** `int`	**From** `long`	**From** `uint`	**From** `ulong`
To `short`	X	ex	ex	ex	ex
To `int`	im	X	ex	ex	ex
To `long`	im	im	X	im	ex
To `uint`	ex	ex	ex	X	im
To `ulong`	ex	ex	ex	im	X

Remember when you read these tables that you must take into account that `long` means a different thing in C# than in C++. In C++/CLI, both `long` and `int` are aliases for `System::Int32`, and `long long` is used for `System::Int64`, whereas C# achieves it by using `long`. Table 11-3 is an excerpt from the type table included in Chapter 6.

Table 11-3. *A Partial Type Table*

C#	C++/CLI	Size	Type	Signed	Marshal	Example
`short`	`short`	2	Int16	Yes	No	`-1`
`ushort`	`unsigned short`	2	UInt16	No	No	`3u`
`int`	`int`	4	Int32	Yes	No	`-1l`
`uint`	`unsigned int`	4	UInt32	No	No	`3ul`
`long`	`long long`	8	Int64	Yes	No	`3ll`
`ulong`	`unsigned long long`	8	UInt64	No	No	`3ull`
`float`	`float`	4	Single	Yes	No	`4.0f`
`double`	`double`	8	Double	Yes	No	`3.0`

Now let's look at the conversion tables. You may notice in the C# table that there is no possible loss of data. I'm a C++ advocate, but I must admit that I prefer the C# implementation in this area. Notice also that C++/CLI also considers implicit every conversion that C# considers implicit but still reports a signed/unsigned mismatch when extending from `unsigned int` to `long`.

The good news is that, when we look at these tables, we discover that C++/CLI has a way to get the same warning levels you've come to expect in C#. It's unintuitive but simple—don't use `int`.

In C++/CLI, `int` and `long` both map to `System::Int32`, and `unsigned int` and `unsigned long` both map to `System::UInt32`, but they are treated differently for warning purposes by the C++ compiler. Much of the reason for this is historical; `short` and `long` were initially defined as the minimum and maximum integer sizes supported by the target architecture. The type `int` was defined as the most efficient size for the target architecture. Over time, implementations found that this sort of floating definition made porting programs between platforms problematic. This led to the current implementation on .NET which fixes `short` at 16 bits, and `int` and `long` at 32 bits. The type `long long` was added to the language for 64 bits for .NET.

For other target architectures, `int` is either implemented as a `short` or a `long`, making the issuance of warnings problematic. The unofficial programming practice was to use `int` when you didn't really care about conversion problems and needed fast, efficient code; if the data itself called for it, you'd use `short` or `long`. This practice still applies today in .NET: use `short` and `long` over `int`, and the compiler will do its part and issue warnings.

Now let's have a look at floating point conversions.

Floating Point Conversion Tables

Let's examine some cross conversions in C++ in Table 11-4.

Table 11-4. *C++ Conversion Table for Floating Point Types and for a Sampling of Integer Types*

C++		From	
To	**From** `int`	**From** `float`	**From** `double`
To `int`	X	i2	i2
To `float`	i2	X	i3
To `double`	im	im	X

Let's examine some cross conversions in C# in Table 11-5.

Table 11-5. *C# Conversion Table for Floating Point Types and for a Sampling of Integer Types*

C++		From	
To	**From** `int`	**From** `float`	**From** `double`
To `int`	X	ex	ex
To `float`	im	X	ex
To `double`	im	im	X

One interesting thing to note about the floating point conversion tables is that C++ issues a level 2 warning when promoting an `int` to a `float` but not when promoting an `int` to a `double`. Consider the following snippet:

```
using namespace System;
void main()
{
    int i0 = int::MaxValue;
    int i;
    float f;
    double d;
    f = i0;
    i = f;
    Console::WriteLine("int {0}, to float {1}, back to int {2}", i0, f, i);
    d = i0;
    i = d;
    Console::WriteLine("int {0}, to double {1}, back to int {2}", i0, d, i);
}
```

In this example, we take the maximum positive integer and convert it to a `float` and back, with loss of data. Even though `int` and `float` are both 4 bytes long, `float` uses some of those bits to store exponent and sign information, so it is not able to store the integer information with full precision. If you take a similar round-trip using `double`, there is no data loss. Thus, the warning is correct:

```
C:\>cl /clr:pure /nologo /W4 test.cpp
test.cpp(8) : warning C4244: '=' : conversion from 'int' to 'float', possible loss
of data
test.cpp(9) : warning C4244: '=' : conversion from 'float' to 'int', possible loss
of data
test.cpp(12) : warning C4244: '=' : conversion from 'double' to 'int', possible loss
of data
C:\>test
int 2147483647, to float 2.147484E+09, back to int -2147483648
int 2147483647, to double 2147483647, back to int 2147483647
```

You might think there is a problem with the C# compiler, as it allows the conversion from `int` to `float` without warning about possible loss of data. However, C# requires the conversion back to `int` from `float` to have an explicit conversion or cast, so you can argue that a warning on the outbound direction is extraneous. The C# code follows:

```
using System;
class R
{
    public static void Main()
    {
        int i0 = int.MaxValue;
        int i;
        float f;
        double d;
        f = i0;
        i = (int)f;
        Console.WriteLine("int {0}, to float {1}, back to int {2}", i0, f, i);
        d = i0;
        i = (int)d;
        Console.WriteLine("int {0}, to double {1}, back to int {2}", i0, d, i);
    }
}
```

The results for the C# version follow:

```
C:\>csc /nologo test.cs
C:\>test
int 2147483647, to float 2.147484E+09, back to int -2147483648
int 2147483647, to double 2147483647, back to int 2147483647
```

Note that the C# version required explicit conversions in order to compile.

User-Defined Conversions

In the same way that the compiler defines implicit and explicit conversions between built-in types, users can define implicit and explicit conversions between user-defined types. In C#, you use the `implicit` and `explicit` keywords. In C++/CLI, conversions are implicit by default, and you use the `explicit` keyword to specify an explicit conversion.

Implicit Conversions

In our `Complex` class, we used a private helper function to define division of a complex number by a `double`. Why not expose this? Beyond that, why not allow users to multiply a complex number by a `double` or a `double` by a complex number?

We could write specific overloads for each of these operations, or we could define an implicit operator that converts a `double` to a `Complex`. Here it is, in C++/CLI syntax; it is a static member function that takes a `double` parameter:

```
static operator Complex(double re)
{
    return Complex(re,0);
}
```

Now users can perform all of the basic mathematical operations on complex numbers and doubles. Here is a new version of `main()` that uses this implicit operator:

```
void main()
{
    Complex a(-5,10), b(3,4);
    double c(3.5);
    Console::WriteLine("({0}) / ({1}) = {2}",a,b,a/b);
    Console::WriteLine("({0}) * ({1}) = {2}",a,c,a*c);
    Console::WriteLine("({0}) / ({1}) = {2}",c,a,c/a);
}
```

After compiling and running, we get the following:

```
C:\>cl /clr:pure /nologo test.cpp
C:\>test
(-5 + 10i) / (3 + 4i) = 1 + 2i
(-5 + 10i) * (3.5) = -17.5 + 35i
(3.5) / (-5 + 10i) = -0.14 + -0.28i
```

That's a lot of power for a little bit of work. What about going the other direction, from a `Complex` to a `double`?

Explicit Conversions

Going from a `Complex` to a `double` is going to involve some data loss, so this should not be an implicit conversion. What might this mean? Should we project the complex number onto the real line, and just return the real part of the complex number? Should we return the magnitude of the complex number? Should a conversion to `double` even exist? It's really up to us to decide which way to go.

Personally, I am drawn to the idea of using magnitude:

$$|a+bi| = \sqrt{a^2+b^2}$$

The explicit conversion looks the same as the implicit conversion, except for the `explicit` keyword:

```
static explicit operator double(Complex c)
{
    return Math::Sqrt(c.re*c.re + c.im * c.im);
}
```

Just for neatness, we can replace the following code:

```
static Complex operator / (Complex a, Complex b)
{
    return a / (b.re*b.re+b.im*b.im) * ~b;
}
```

with

```
static Complex operator / (Complex a, Complex b)
{
    return a / ((double)b * (double)b) * ~b;
}
```

This gives us the following finished program; note that `operator/(Complex, double)` is no longer `private`:

```
using namespace System;
value struct Complex
{
    double re;
    double im;
    Complex(double re, double im)
    {
        this->re = re;
        this->im = im;
    }
    static Complex operator + (Complex a, Complex b)
    {
        return Complex(a.re+b.re, a.im+b.im);
    }
    static Complex operator - (Complex a, Complex b)
    {
        return Complex(a.re-b.re, a.im-b.im);
    }
    static Complex operator ~ (Complex a)
    {
        return Complex(a.re, - a.im);
    }
```

```
    static Complex operator * (Complex a, Complex b)
    {
        return Complex(a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re);
    }
    virtual String ^ ToString() override
    {
        String ^s = re.ToString();
        if(im != 0)
        {
            return s += " + " + im.ToString() + "i";
        }
        return s;
    }
    static Complex operator / (Complex a, Complex b)
    {
        return a / ((double)b * (double)b) * ~b;
    }
    static operator Complex(double re)
    {
        return Complex(re,0);
    }
    static explicit operator double(Complex c)
    {
        return Math::Sqrt(c.re*c.re + c.im * c.im);
    }
    static Complex operator / (Complex a, double f)
    {
        return Complex(a.re/f, a.im/f);
    }
};
void main()
{
    Complex a(-5,10), b(3,4);
    double c(3.5);
    Console::WriteLine("({0}) / ({1}) = {2}",a,b,a/b);
    Console::WriteLine("({0}) * ({1}) = {2}",a,c,a*c);
    Console::WriteLine("({0}) / ({1}) = {2}",c,a,c/a);
}
```

CLS-Compliant Operators

There are several ways to define the operators in C++, depending on your intentions and goals. In this case, the goal is to create a CLS-compliant application. The Common Language Specification (CLS) defines how code may be made compatible with multiple CLI languages. Therefore, it is best to adopt this paradigm when writing programs that link with C# or other .NET languages. An operator is said to be CLS-compliant when all of the following criteria are met:

- The operator is listed in the CLS-compliant tables, as stated in the CLS.
- The operator is a static member of a reference or value class.
- Parameters and return values of the operator function are not passed or returned by any pointer, reference, or handle.

Let's examine the CLS-compliant unary operators in Table 11-6.[4]

Table 11-6. *CLS-Compliant Unary Operators*

Operator Name	Function Name	C#	C++
operator&	AddressOf	No	Yes
operator!	LogicalNot	Yes	Yes
operator~	OnesComplement	Yes	Yes
operator*	PointerDereference	No	Yes
operator-	UnaryNegation	Yes	Yes
operator+	UnaryPlus	Yes	Yes
operator true	true	Yes	No
operator false	false	Yes	No

Let's examine the CLS-compliant binary operators in Table 11-7.

Table 11-7. *CLS-Compliant Binary Operators*

Operator Name	Function Name	C#	C++
operator+	Addition	Yes	Yes
operator&	BitwiseAnd	Yes	Yes
operator|	BitwiseOr	Yes	Yes
operator,	Comma	Yes	Yes
operator--	Decrement	Yes	Yes
operator/	Division	Yes	Yes
operator==	Equality	Yes	Yes
operator^	ExclusiveOr	Yes	Yes
operator>	GreaterThan	Yes	Yes
operator>=	GreaterThanOrEqual	Yes	Yes
operator++	Increment	Yes	Yes
operator!=	Inequality	Yes	Yes
operator<<	LeftShift	Yes	Yes
operator<	LessThan	Yes	Yes
operator<=	LessThanOrEqual	Yes	Yes
operator&&	LogicalAnd	No	Yes

(*continued*)

[4]These tables are from the C++/CLI Language Specification.

Table 11-7. (*continued*)

Operator Name	Function Name	C#	C++
operator||	LogicalOr	No	Yes
operator%	Modulus	Yes	Yes
operator*	Multiply	Yes	Yes
operator>>	RightShift	Yes	Yes
operator-	Subtraction	Yes	Yes

Most of these operators are fairly self-explanatory. Only a few merit special mention:

- operator* can either be Multiply or PointerDereference, depending on whether it is a binary or unary operator.
- operator& can either be BitwiseAnd or AddressOf, depending on whether it is a binary or unary operator.
- Both operator&& and operator|| can be overloaded in C++/CLI. These cannot be overloaded in C#.
- operator true and operator false are not implemented in C++.

operator true and operator false

operator true and operator false are used in the following C# code, which cannot be written similarly in C++/CLI:

```
using System;
class R
{
    int value;
    R(int V)
    {
        value = V;
    }
    public static bool operator true (R r)
    {
        return r.value!=0;
    }
    public static bool operator false ( R r)
    {
        return r.value==0;
    }
    public void Test(String name)
    {
        if(this)
        {
            Console.WriteLine("{0} is true", name);
        }
```

```
        else
        {
            Console.WriteLine("{0} is false", name);
        }
    }
    public static void Main()
    {
        R r3 = new R(3);
        r3.Test("r3");
        R r0 = new R(0);
        r0.Test("r0");
    }
}
```

If you compile and run this in C#, you get the following results:

```
C:\>csc /nologo test.cs
C:\>test
r3 is true
r0 is false
```

There is a nice workaround in C++ using an implicit conversion to `bool`:

```
using namespace System;
ref class R
{
private:
    int value;
    R(int V)
    {
        value = V;
    }
public:
    static operator bool(R^ r)
    {
        return r->value != 0;
    }
    void Test(String^ name)
    {
        if(this)
        {
            Console::WriteLine("{0} is true", name);
        }
        else
        {
            Console::WriteLine("{0} is false", name);
        }
    }
```

```
    static void Main()
    {
        R ^r3 = gcnew R(3);
        r3->Test("r3");
        R ^r0 = gcnew R(0);
        r0->Test("r0");
    }
};
void main()
{
    R::Main();
}
```

Other Operators

C++ allows you to overload the assignment operators, function calls (`operator()`), and indices (`operator[]`) in a manner that is not CLS compliant.

Summary

Having a good feel for expressions and operators is essential in object-oriented programming. They allow you to extend your classes and work with them as if they were built-in types.

In the next chapter, we'll complete our tour of basic C++ by filling in some of the details that have not been covered in any of the broad categories of previous chapters. After that, you will be well prepared to go into greater depth and detail in the chapters that follow.