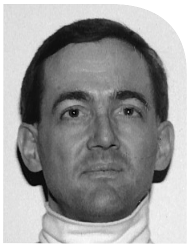
;login:

THE MAGAZINE OF USENIX & SAGE
June 2003 • volume 28 • number 3

inside:

C# types

by Glen McCluskey



Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

In our examination thus far of the C# language, we've looked at the overall architecture and also discussed some basics of compiling and executing programs. In this column we'll start to consider the various types that C# offers.

C# is an object-oriented language, and its type system thus centers around system-provided and user-defined classes, classes that represent an abstraction of some sort (such as a calendar date or a geometric X,Y point). All the details of how classes work cannot be covered in a single column, and we'll touch on them only briefly in this initial presentation.

Built-in Types

C# offers a standard set of built-in data types, such as short, long, and double. These are similar to the corresponding types in C/C++. They have a standard size; for example, long is 64 bits. Values of the char type hold a single Unicode character (16 bits); an 8-bit unsigned byte type exists as well.

One substantial difference from C is the provision of a decimal type, in addition to the usual floating-point types float and double. Floating-point arithmetic does poorly at handling decimal calculations (e.g., payroll deductions), and the decimal type offers an alternative. Here's an example of where it matters:

```
using System;

public class Dec {
      public static void Main() {

          // add 0.1 to itself twice and compare to 0.3,
          // using the double type

          double dbl = 0.1;
          if (dbl + dbl + dbl == 0.3)
              Console.WriteLine("double is equal");
          else
              Console.WriteLine("double is unequal");

          // the same, but using the decimal type

          decimal dec = 0.1m;
          if (dec + dec + dec == 0.3m)
              Console.WriteLine("decimal is equal");
          else
              Console.WriteLine("decimal is unequal");
      }
}
```

Using a double type, 0.1 added to itself three times does not result in 0.3, due to floating-point representation problems: 0.1 is the sum of an infinite series of negative powers of two (0.00110011001...) and therefore is not exactly representable in floating-point format. The decimal type solves this problem and is useful in areas such as financial calculations. Of course, it's a bit slower in execution.

Another difference from C is the string type. Here's an example:

```
using System;

public class String {
      public static void Main() {
          string s = "testing";
          Console.WriteLine(s);
      }
}
```

A rich set of functions for manipulating strings is also available.

Type-Checking and Conversions

C# applies tighter type-checking rules to the use of built-in types than C and C++. For example, in this code:

```
using System;

public class Conv {
      public static void Main() {
          ulong a = 0xffffffff;
          uint b;

          //b = a;
          b = (uint)a;
      }
}
```

an explicit cast is required to convert the unsigned long value to unsigned int. Such conversion often represents a programming mistake, and the programmer is required to specify explicitly that the conversion is desired (and, presumably, to consider its implications).

A related example is the use of Boolean expressions:

```
using System;

public class Bool {
      public static void Main() {
          int x = 100;
```

```
            //if (x) {}
            if (x != 0) {}
        }
    }
```

C# requires that the controlling expression in an if statement be of Boolean type, not simply a numeric or pointer type that can be checked against zero.

Boxing and Unboxing

What is the relation of built-in types like int to class types? In some languages, there is no connection – the two kinds of types have nothing in common.

C# approaches things a little differently. An automatic conversion called "boxing" is supplied by the compiler in order to convert a value of a built-in type to a class type. Let's look at an example:

```
using System;
using System.Collections;

public class Box {
    public static void Main() {

        // box an integer value

        int n = 100;
        object obj = n;

        // unbox it

        //n = obj;
        n = (int)obj;

        // add some integer values to a collection

        ArrayList list = new ArrayList();
        list.Add(1);
        list.Add(2);
        list.Add(3);
        for (int i = 0; i < list.Count; i++)
            Console.WriteLine(list[i]);
    }
}
```

In the first part of the code, an object of the root class System.Object is initialized with an integer value (100). The conversion involves creating a wrapper object of class type System.Int32 for the integer value. In other words, an object of the class System.Int32 is created and initialized with the value 100. Because this data type is part of the class hierarchy with System.Object as its root, the assignment is valid.

Here's some intermediate language output that illustrates what is going on in the first part of the example:

```
IL_0000:  ldc.i4.s      100
IL_0002:  stloc.0
IL_0003:  ldloc.0
IL_0004:  box           [mscorlib]System.Int32
IL_0009:  stloc.1
IL_000a:  ret
```

The boxing conversion is not without cost, but at the same time eases programming. In the second part of the example, some values of a built-in type are added to an ArrayList collection. The collection requires values of "object" type, so the integers are boxed automatically before being added to the collection.

The automatic boxing conversion, along with the existence of wrapper classes such as System.Int32, implies that the distinction between built-in and class types is blurred to some extent.

Enumerated Types

C# also supports enumerated types, similar to those offered by C. Here's some code that defines an enum to represent the colors red, green, and blue:

```
using System;

enum Color : byte {RED = 1, GREEN = 2, BLUE = 3}

public class Enum {
    public static void Main() {
        Color c = Color.GREEN;

        //int i = c;
        int i = (int)c;

        Console.WriteLine(i);
    }
}
```

The enum base type is byte, an unsigned value 0–255. Enum types are not interchangeable with integral types; conversion requires an explicit cast.

Class and Struct Types

The C# concept of a class is similar to that found in the C++ and Java languages. A class defines some data (which objects of the class will contain) and operations on that data, expressed through functions ("methods") of the class. If you're a C programmer, a class is a grouping of some data defined in a struct coupled with some functions that operate on instances of the struct. Data in objects is generally hidden or private, and accessible only via the methods of the object's class.

Let's look at an example, one that uses a class to represent X,Y points:

```
using System;

public class PointClass {
      private int x, y;

      public PointClass(int x, int y) {
          this.x = x;
          this.y = y;
      }
      public int getX() {return x;}
      public int getY() {return y;}
}

public struct PointStruct {
      private int x, y;

      public PointStruct(int x, int y) {
          this.x = x;
          this.y = y;
      }

      public int getX() {return x;}
      public int getY() {return y;}
}

public class Struct {
      public static void Main() {
          PointClass pc = new PointClass(10, 20);
          PointStruct ps = new PointStruct(30, 40);
      }
}
```

The PointClass class defines private data members x and y, along with a constructor to create new objects (a constructor is denoted as a method with the same name as the class). There are also getX and getY accessor methods, to access the x and y values from a PointClass object.

In the Main method, an object of PointClass is created using the new operator, and the constructor is called after space is allocated from the heap. C# uses garbage collection, so you don't need to worry about explicitly freeing object space when you're done with it; garbage collection takes care of this for you automatically.

PointStruct is a class that seems identical to PointClass, except that it's defined with a struct keyword instead of a class. What's the difference? A struct is a lighter-weight type than a class, with some restrictions. For example, a struct cannot inherit from another class or struct.

A really key difference is that a struct is a value type, whereas a class is a reference type. In the example above, when an object of PointStruct is allocated via the new operator, it's allocated from the stack, not the heap. Here's some intermediate language output that shows the difference between the two new calls:

```
IL_0000:  ldc.i4.s     10
IL_0002:  ldc.i4.s     20
IL_0004:  newobj       instance void PointClass::.ctor
IL_0009:  stloc.0
IL_000a:  ldloca.s     V_1
IL_000c:  ldc.i4.s     30
IL_000e:  ldc.i4.s     40
IL_0010:  call         instance void PointStruct::.ctor
IL_0015:  ret
```

When struct objects are passed to methods, they are passed by value, with a copy made of the object. For example, in this code:

```
using System;

public class A {            // class
      public int x;
}

public struct B {           // struct
      public int x;
}

public class Struct2 {
      public static void f(A aref, B bref) {
          aref.x = 30;
          bref.x = 40;
      }

      public static void Main() {
          A aref = new A();
          aref.x = 10;

          B bref = new B();
          bref.x = 20;

          f(aref, bref);

          Console.WriteLine("{0} {1}", aref.x, bref.x);
      }
}
```

the values "30 20" are printed. The object of class A is passed by reference, whereas the B object is passed by value. Method f can modify the value of the A object in a way that's visible outside of f, because it has a pointer to the object. But f has only a copy of the B object.

A struct is most useful for types that are very simple. If you're defining an X,Y point type, for example, it might be worth using a struct instead of a class.

In future discussions, we'll get into more detail of how classes and structs work and look at related concepts such as interfaces and abstract classes.