;login:

THE MAGAZINE OF USENIX & SAGE

February 2003 • volume 28 • number 1

inside:

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

an introduction to C#

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

C# (pronounced "cee sharp") is a new programming language, part of the .NET Framework initiative from Microsoft. This column is the first of a series that will discuss the C# language and libraries. But before we delve into technical details, we need to present a little background and show where C# fits into the larger picture.

A Proprietary Language?

An obvious question about C# is whether it is simply another in a series of proprietary languages (e.g., Visual Basic). Such languages are clearly useful but live in a different world from standardized languages like C. It's impossible to predict the future with any certainty, but C# does have a shot at becoming a widely used standard. The language has a specification external to Microsoft, and several independent projects are underway to develop C# compilers: for example, the Mono effort for Linux. We will give details of these efforts later in the discussion.

The .NET Framework

If you read the technical press at all, you've probably heard of something called the .NET Framework. This is an elusive term. What does it mean? One way of illustrating the concepts of the .NET Framework is to consider what happens when a C# program is compiled and executed. Let's start with the Hello program:

```
using System;

class Hello {
      static void Main() {
          Console.WriteLine("Hello, World!");
      }
}
```

The first interesting part of .NET is compilation. This program is compiled into an intermediate language called MSIL or IL, not straight into binary code. If I'd written the same program in a different language supported by .NET, the IL representation would be similar to what is produced for the C# program above. This point illustrates one of the key goals of the .NET effort – the ability to mix code written using different programming languages. This goal is supported by a common intermediate language.

The IL output for the Hello program looks like this:

```
.method private hidebysig static void Main() cil managed
{
      .entrypoint
      // Code size    11 (0xb)
      .maxstack 1
      IL_0000: ldstr "Hello, World!"
      IL_0005: call void [mscorlib]System.Console::
                  WriteLine(string)
      IL_000a: ret
} // end of method Hello::Main
```

The program is compiled into an intermediate language which is then executed at some later time. What happens then? A piece of the .NET Framework called the Common Language Runtime (CLR) actually executes the intermediate form of the program. The intermediate is compiled on demand into machine code, using a just-in-time compiler (JIT), and executed. The CLR also takes care of issues such as memory layout, garbage collection, and security. Furthermore, it provides a certain execution environment paradigm that supports the languages available for .NET.

If you study this example a bit, it's obvious that the code is making reference to some sort of a standard library – note the mention of System and Console.WriteLine, and so on. Another part of the .NET Framework is a set of framework base classes, used for performing operations such as I/O, string manipulation, and networking. The Hello program makes use of some of these classes for actual output to the console.

As we already mentioned, it's possible to mix languages and libraries within .NET. So in our Hello example, it's possible that the Console.WriteLine method for doing I/O is not in fact written in C# – it may be implemented in some other language. A couple of pieces of the .NET Framework called the Common Language Specification and Common Type System are used to support such interoperability. These specifications describe areas such as inheritance, object properties, exceptions, interfaces, and values. This whole area is in some sense analogous to the older terms "calling conventions" and "runtime environment" that we've always had to worry about when mixing languages. For example, if I have some C++ code, and I call a C function, compiled with a different compiler, then I need to

worry about such things as whether function arguments are pushed onto the stack from left to right or right to left. I need to be concerned with whether two different languages that I'm working with use the same byte order and size and representation for data.

Higher-Level Services

In discussing the .NET Framework, thus far we've looked at low-level features. There's the Common Language Runtime, along with standards such as the Common Type System and the Common Language Specification that describe how languages interoperate. There is also a set of framework base classes that is part of this core package.

The .NET Framework also contains several groups of higher-level services and classes:

- ADO.NET supports database manipulation and XML data handling.
- Windows Forms are the basic mechanism for building windows-based applications. A Forms object represents windows in your application.
- Web Forms are a mechanism for dynamically generating Web pages on a server, combining a static HTML page with C# code that generates dynamic content. The C# code runs on a server with the resulting generated HTML page being sent to a Web browser. Web Forms are something like Active Server Pages.
- Web Services allow you to build components whose methods can be invoked across the Internet. It is based on SOAP (Simple Object Access Protocol), which in turn is based on XML, HTTP, and SMTP.

You can also combine C# code with code written in other .NET languages, including VB.NET, Managed C++, and JScript .NET.

The .NET Framework Hierarchy

If we represent the .NET Framework using a hierarchy from highest to lowest levels, it would look something like this:

C#, VB.NET, Managed C++, JScript .NET
Windows Forms, Web Forms, Web Services
ADO.NET, XML
Framework Base Classes (I/O, string, networking, etc.)
Common Language Runtime
  Memory Layout
  Garbage Collection
  Security
  Debugging
  Exception Handling
  Just-in-Time Compilation
Common Type System, Common Language Specification
Operating System

The Flavor of C#

What is C# like? Let's look at a few key areas to help answer this question.

C# is an object-oriented language, similar in many ways to the C++ and Java languages. It also uses syntax similar to what you're already familiar with in C or C++. For example, this program adds two numbers and prints the sum:

```
using System;

class prog1 {
     static int add(int a, int b) {
        return a + b;
     }

     static void Main() {
        int a = 37;
        int b = 47;

        int c = add(a, b);

        Console.WriteLine("{0}", c);
     }
}
```

C# is a "safe" language, meaning that common problems such as memory leaks, de-referencing invalid pointers, or ignoring error return codes are much less of an issue than with other languages. C# uses garbage collection instead of user-level memory management, doesn't normally allow the use of pointers, and uses exceptions to propagate errors. If you need to use pointers, you can explicitly do so by means of an "unsafe" method modifier that allows pointers within that method. For example:

```
using System;

class prog2 {
     unsafe static void Main() {
        char* p = (char*)0x1234;
        *p = 'x';
     }
}
```

C# supports attributes and metadata and reflection. For example, you can devise custom attributes and use them to represent detailed information about bug fixes you have made in your code. Such information could also be represented within comments, which is a traditional approach, but attributes have a major advantage – they are not unstructured comments but can be queried by a C# program. They are data about your code that is carried along with your code.

C# is Internet-centric. For example, it includes support for remote method invocation, XML and XML documentation

comments, serializing objects to be sent across a network, and so on.

In this column we've described the context in which C# operates. In future columns we'll start looking at the language itself, and examine some of its distinctive features.

References

Many C# books are available. Two recommended ones are:

Jesse Liberty, *Programming C#*, 2d ed. (Sebastopol, CA: O'Reilly, 2002).

Eric Gunnerson, *A Programmer's Introduction to C#*, 2d ed. (Berkeley, CA: Apress, 2001).

C# Compilers and Development Environments

Here are Web links for three different C# compilers you can download. The first two of these are independent efforts, and the last is the Microsoft SDK:

http://www.go-mono.com/c-sharp.html

http://www.southern-storm.com.au/portable_net.html

http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml

C# Standardization

The C# language has an external specification, found at the European Computer Manufacturers Association (ECMA) Web site: *ftp://ftp.ecma.ch/ecma-st/Ecma-334.pdf*.