CHAPTER 1

Hello, World!

THIS CHAPTER IS YOUR starting point in learning MC++. Here you will see what managed code is, and how to write a simple managed program and a simple managed class that can be exposed to other languages.

Your First Program

When learning a new programming language, the first program a programmer sees often does nothing but print the phrase, "Hello, World!" We will follow the same tradition here—even though, strictly speaking, Managed C++ cannot be called a *new* language, being based as much as it is on C++.

Because any valid C++ program is a valid MC++ program, we will start with a simple C++ example and turn it into managed C++ code. Let's begin by creating a file called `Hello.cpp` with the following text:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!";
}
```

The same compiler is used to compile both unmanaged and managed C++ code. If you are working at the command line, use the `/clr` compiler option to tell the compiler you want your code to be managed. When working with the Visual Studio .NET integrated environment, enable the Use Managed Extensions property in the General property page of the project's Properties dialog box.

To compile `Hello.cpp` from the command line, enter the following:

```
cl hello.cpp /clr /EHsc
```

The compiler will create an executable file called `hello.exe`. When you run it, you will see that it does what you would expect it to do—print a message on the screen:

```
Hello, World!
```

That's it! You have created your first managed program—although it doesn't really use any functionality provided by the .NET Framework. Your first *truly* managed program would look like the following:

```
#using <mscorlib.dll>
using namespace System;
int main()
{
    Console::WriteLine( S"Hello, World!" );
}
```

The first statement of the program, the preprocessor directive `#using <mscorlib.dll>`, tells the compiler to find and load the file `mscorlib.dll`, which contains the .NET *base class library* (BCL). The directive `#using` is similar to the C++ `#include` directive except that the imported file contains definitions of the types and modules in special binary format that is available to programs written in any .NET language. Such files are called *assemblies* and *modules* (you can read more about these files in Chapter 2).

Similar to classes defined in the Standard C++ Library, BCL classes are enclosed in namespaces. The `using namespace` directive brings all symbols of the specified namespace into the current scope, so you can write `Console::WriteLine` instead of `System::Console::WriteLine`.

One of those symbols, class `Console`, provides functionality for basic textual input-output. Method `WriteLine` prints out a string and moves the caret position down to the next line.

`Console::WriteLine` expects a pointer to the BCL class `System::String` as an argument. The compiler allows you to create instances of this class by preceding a string literal with the prefix `S`—similar to how you would create a wide string with the prefix `L`.

Managed versus Unmanaged Code

What does it really mean for the code to be managed? The executable file produced by the MC++ compiler does not contain any x86 instructions, except for the startup stub. Instead, the code is compiled into an intermediate language called MSIL, which stands for Microsoft Intermediate Language (see Figure 1-1), and the *metadata*, which records the information about all objects that compose the module.

MSIL is then compiled into the native code at runtime by the Just In Time (JIT) compiler every time the program is executed. Alternatively, MSIL can be precompiled by JIT at the program installation time.

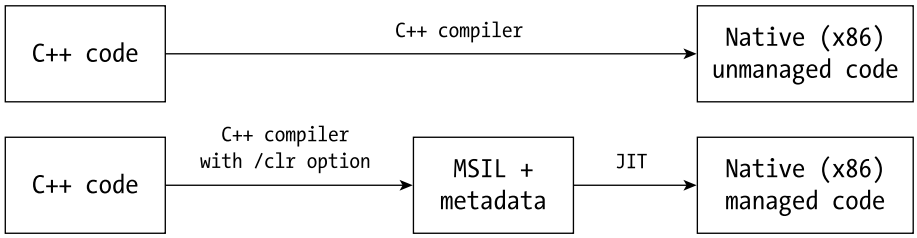

Figure 1-1. Managed code versus unmanaged code

The code produced by the JIT compiler (either at runtime or installation time) is managed by the .NET execution environment, which tracks pointers into the managed heap (to make automatic garbage collection possible), and provides security checks and other services.

Unmanaged code, however, does not and cannot use any services provided by the .NET Framework. Yet its advantage is the absence of the performance overhead imposed by the execution environment.

While developing the MC++ compiler, substantial effort was devoted to ensuring that all existing C++ programs work correctly when they are compiled into MSIL. This requirement was called IJW—short for "It Just Works." IJW is a fundamental necessity for enabling porting of existing C++ programs to .NET. Thanks to IJW, you don't have to change your code at all; compile it with the `/clr` compiler option, and it will "just work."

Even though the MC++ compiler tries to compile every C++ program into MSIL, it is not always possible to do this. For instance, a function containing inline x86 assembler instructions cannot be compiled into MSIL, simply because MSIL is a platform-independent language of higher level than the x86 assembler. Such a function has to be compiled into unmanaged code; it cannot use any functionality provided by the .NET Framework.

Fortunately, both managed and unmanaged code can coexist in the same compilation unit. This means you can create managed and unmanaged functions in the same source file. You can call an unmanaged function from a managed function, which is why it is possible to call C runtime functions, such as `printf`, or WinAPI functions, such as `CreateWindow`, from managed code.

Interoperability

Let's see how things work across different languages. We will create a simple MC++ function that generates Fibonacci numbers and call it from a C# program. Note that in C# all functions must be members of a class.

```
#using <mscorlib.dll>
public __gc class Fibonacci
{
public:
    static int GetNumber( int n )
    {
        int previous = -1;
        int result = 1;
        for( int i=0; i<=n; i++ )
        {
            int sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }
};
```

This program defines a managed class, `Fibonacci`, with a static method, `GetNumber`, that returns the *n*th Fibonacci number. Managed classes are described in detail in Chapter 3, but for now just remember that managed classes can be exposed to other languages.

As you might have noticed, there is no `main` function in this program. This is because the program is a library, not a standalone application; its sole purpose is to expose the functionality provided by the class `Fibonacci`. Usually, programs like this are compiled into DLLs, not executable files. Here is how to do so: call the source file `Fibonacci.cpp` and compile it like this:

```
cl Fibonacci.cpp /clr /LD
```

The resulting binary file will be called `Fibonacci.dll`.

Now create a C# file, `TestFibonacci.cs`:

```
using System;
class TestFibonacci
{
    public static void Main()
    {
        Console.WriteLine( "Fibonacci number 10 is {0}",
                                                Fibonacci.GetNumber(10) );
    }
}
```

Compile this C# program with a reference to the assembly `Fibonacci.dll`:

```
csc TestFibonacci.cs /reference:Fibonacci.dll
```

When executed, the resulting program `Fibonacci.exe` prints:

```
Fibonacci number 10 is 55
```

As you can see, it is easy to use a MC++ class in C#. It is also easy to consume an assembly in MC++. The following program imports `Fibonacci.dll` (the same one created before or written in some other language):

```
#using <mscorlib.dll>
using namespace System;
#using <Fibonacci.dll>
int main()
{
    Console::WriteLine( S"Fibonacci number 10 is {0}",
        Fibonacci::GetNumber(10).ToString() );
}
```

Writing a multilanguage application is easy—so is debugging. You will appreciate a feature of Visual Studio .NET debugger that allows you to step from a source written in one language to a source written in another language.

Summary

Managed code and interoperability are the cornerstone concepts of MC++, and as such are the main focus of this book. In this chapter, we looked at how to write managed code that can be consumed by programs written in other .NET languages.

In the next chapter, we will take a deeper look at the .NET Framework—the environment in which all managed programs operate.