Appendix G: Writing Managed C++ Code for the .NET Framework

What Is .NET?

.NET is a powerful object-oriented computing platform designed by Microsoft. In addition to providing traditional software development tools, it provides technologies to create Internet-based programs, and programs that provide services over the Web.

.NET consists of several layers of software that sit above the operating system and provide a managed environment in which programs can execute. Within this environment, .NET manages a program's memory allocation and destruction needs, resource usage, and security. For software developers, .NET consists of the following important components:

- The Common Language Runtime (CLR)
- The Common Type System (CTS)
- The .NET Framework Class Library

Let's briefly look at each of these.

The Common Language Runtime

The *Common Language Runtime*, or *CLR*, is the part of .NET that actually runs application code. The CLR is a *managed runtime environment*. This means that the CLR executes code within an environment where memory allocation and de-allocation, security, and type compatibility are strictly managed.

The Common Type System

The *Common Type System* or *CTS* is a set of common data types provided by .NET. These data types are available to all applications running in the CLR, and they have the same characteristics regardless of the programming language used.

In C++, the standard primitive data types all correspond directly to a .NET type. For example, .NET provides a data type named `Int32`, which is a 32-bit integer. In C++, the `int` data type corresponds to `Int32`. Table G-1 shows a list of the .NET common types, as well as the C++ types that they correspond to. Note that C++ does not provide primitive types for all of the .NET types.

Table G-1 .NET Common Types

.NET Common Type	Description	Equivalent C++ Type
`Byte`	8-bit unsigned integer	`char`
`SByte`	8 bit signed integer	`signed char`
`Boolean`	8-bit Boolean value (`true` or `false`)	`bool`
`Char`	16-bit Unicode character	`wchar_t`
`Int16`	16-bit signed integer	`short`
`Int32`	32-bit signed integer	`int` `long`
`Int64`	64-bit signed integer	`_int64`
`UInt16`	16-bit unsigned integer	`unsigned short`
`UInt32`	32-bit unsigned integer	`unsigned int` `unsigned long`
`UInt64`	64-bit unsigned integer	`unsigned _int64`
`Single`	32-bit single precision floating point	`float`
`Double`	64-bit double precision floating point	`double`
`Decimal`	96-bit floating point value with 28 significant digits	None
`IntPtr`	A signed integer used as a pointer. The size is platform-specific	None
`UIntPtr`	An unsigned integer used as a pointer. The size is platform-specific	None

The .NET Framework Class Library

The .NET Framework Class Library is a library of object-oriented types, such as classes and structures, which provide access to the capabilities of .NET. Programmers may use these object-oriented types to create managed applications that run within the CLR. In

addition, many of the .NET Framework classes may be used as base classes. This allows programmers to create specialized classes that serve specific needs within an application.

Microsoft Intermediate Language

In order for a program to execute, its high-level language source code must be converted to some type of executable code. Traditional compilers translate source code into binary executable code, which may be run directly by the CPU. In order for a program to be run by the CLR, however, it is not compiled to binary executable code. Instead, it must be translated into *Microsoft Intermediate Language*, or *MSIL*. MSIL code is called "intermediate" because it represents an intermediate step between source code and executable code. When a .NET program runs, the CLR reads its MSIL code, then just before execution, converts the MSIL code to binary executable code. The part of the CLR that converts MSIL into binary executable code is called the *Just-In-Time compiler* or *JIT compiler*. Figure G-1 illustrates the process of compiling and executing a .NET program.

NOTE: The JIT compiler doesn't necessarily translate an entire program all at once to executable code. It usually compiles parts of the program as they are needed.

The conversion of source code to MSIL might seem like an unnecessary step, but it has some advantages. First, because all .NET programs are compiled to MSIL, it makes it easier to mix code from several different languages in the same application. Second, because MSIL is not specific to any particular hardware platform, it is portable to any system that supports the CLR.

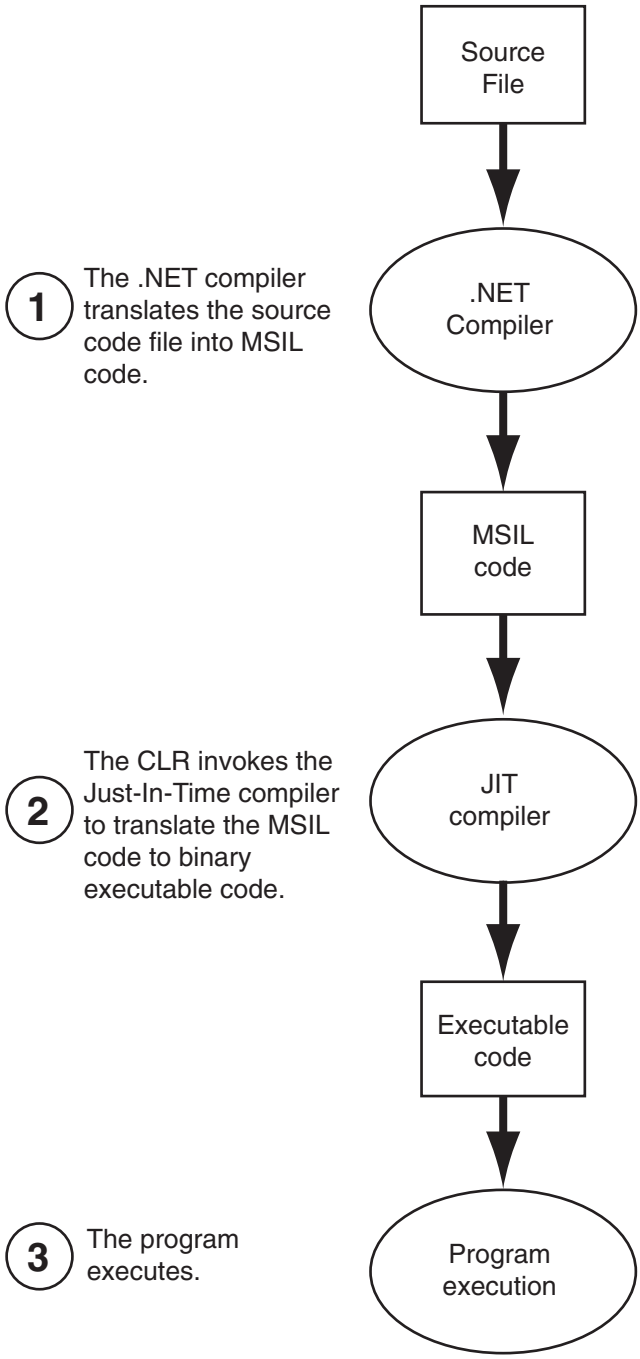
Managed Code

A program may be executed by the CLR if it is written in a language that conforms to the *Common Language Specification*, or *CLS*. The CLS is a set of standards that compiler and programming language designers must follow if they wish for programs written in their language to run in the .NET environment. In Visual Studio .NET, Microsoft provides the C#, Visual Basic .NET, and Visual J# languages, all of which can be used to write CLS compliant code. A Visual C++ compiler is also provided.

However, standard C++ does not produce code that can be managed by the CLR, even if it is written and compiled with Visual C++. To bridge the gap between unmanaged C++ code and the .NET CLR, Microsoft provides extensions to the C++ language which you may use to generate CLS compliant code. C++ code that is written with these extensions is known as *managed* C++ code because it may be executed in and managed by the CLR.

To demonstrate how to write managed C++ code, we will look at two examples: managed dynamic arrays, and managed classes.

Figure G-1



Managed Dynamic Arrays

In standard C++, the programmer must be careful to free the memory used by dynamically allocated objects. For example, look at the following code:

```
const int SIZE = 12;
int *numbers = new int[SIZE];
```

This code dynamically allocates enough memory for an array of twelve integers, and assigns the starting address of the array to the `numbers` pointer. When the program is finished using the array, it should execute the following code to free the memory used by the array:

```
delete [] numbers;
```

In an unmanaged runtime environment, failure to properly reclaim dynamically allocated memory can lead to serious problems. One such problem is known as a *memory leak*. This is when a program repeatedly allocates memory but never frees it. Eventually, the available memory will run out. Another problem occurs when a dynamically allocated object is destroyed, but other code in the program continues to use the object as though it were still in memory.

Because the CLR is a managed runtime environment, it performs automatic *garbage collection*. This means that it automatically frees the memory used by dynamically allocated objects when they are no longer referenced by any part of the program. The programmer no longer has to worry about using `delete` to free memory.

For example, suppose a function in a managed C++ program dynamically allocates an array, and the starting address of the array is stored in a local variable. When the function terminates, the local variable goes out of scope. The dynamically allocated array is no longer referenced by any variables, so the runtime environment will automatically free the memory that it uses.

In Microsoft Visual C++, the syntax for creating a managed dynamic array is different from the syntax that you normally use. Look at the following example:

```
// Create an array of integers.
const int SIZE = 12;
int numbers __gc[] = new int __gc[SIZE];

// Store some values in the array.
for (int i = 0; i < SIZE; i++)
    numbers[i] = i;
```

In this example the third line of code creates a managed array of 12 integers. Notice the use of `__gc` in that line of code. (That's two underscores followed by `gc`.) The `__gc` key word is part of Microsoft's managed extensions for C++. It allows you to dynamically allocate arrays (as well as other objects) which are managed by the CLR. When the array is no longer referenced by any variable, the CLR's garbage collector will free the memory it uses.

NOTE: The garbage collector runs in the background. Under normal circumstances it runs infrequently, allowing more important tasks to operate. If available memory becomes low, however, it will run more frequently. With this in mind, you cannot predict exactly when a dynamically allocated array or object will be deleted.

Managed Classes

In a class declaration you can place the `__gc` key word before the `class` key word to create a managed class. (As previously mentioned, that's two underscores followed by `gc`.) When an instance of a managed class is allocated in memory, the CLR will automatically free it from memory when it is no longer referenced by any variables. Here is an example of a managed class declaration:

```
__gc class Circle
{
private:
   double radius;
public:
   Circle(double rad)
   { radius = rad; }

   double getRadius()
   { return radius; }

   double getArea()
   { return 3.14159 * radius * radius; }
};
```

An instance of the class can then be dynamically allocated in memory, as shown here:

```
Circle *c = new Circle(100.0);
cout << "Radius: " << c->getRadius() << endl;
cout << "Area: " << c->getArea() << endl;
```

When the object is no longer referenced by any variable, it will be freed from memory the next time the the CLR's garbage collection process runs.

When creating managed classes, it is important that you understand the difference between reference types and value types. In standard C++, a class can be used as a *value type*, which means that you can create an instance of the class with a simple declaration statement. A managed class, however, is a *reference type*, which means that you must use the `new` operator to dynamically allocate an instance of the class. For example, had the `Circle` class not been a managed class, we could use the following code to create and use an instance of it.

```
// This code works only with an unmanaged Circle class.
Circle c(100.0);
cout << "Radius: " << c.getRadius() << endl;
cout << "Area: " << c.getArea() << endl;
```

This code will not work, however, with a managed `Circle` class because it is a reference type.

An Example Program

Now that you have an idea of what .NET is, and have been introduced to the concept of managed code, let's look at an example of a managed C++ program. Program G-1 shows a simple "Hello world" program. This code is automatically generated by the Visual Studio .NET application wizard when you create a console application.

Program G-1

```
1  // This is the main project file for VC++ application project
2  // generated using an Application Wizard.
3
4  #include "stdafx.h"
5
6  #using <mscorlib.dll>
7
8  using namespace System;
9
10 int _tmain()
11 {
12    // TODO: Please replace the sample code below with your own.
13    Console::WriteLine(S"Hello World");
14    return 0;
15 }
```

Program Output

```
Hello World
```

Let's take a closer look at this program. First notice that the following directives appear after the comments.

```
#include "stdafx.h"
#using <mscorlib.dll>
```

These `stdafx.h` header file is necessary because it includes all of the other system header files needed for a .NET application. The `#using` directive is used to import an MSIL file into a C++ project. The directive shown here imports the `mscorlib.dll` file, which contains parts of the .NET Framework that you will use the most. For example, all of the CTS data types are defined there.

Next we have the following statement:

```
using namespace System;
```

This statement tells the compiler that we will be using the `System` namespace. The .NET class library is organized into numerous namespaces, and `System` is one of the most commonly used ones. In this namespace are the fundamental components of a .NET application. For example, the names of the CTS data types are part of the `System` namespace. In addition, there are numerous other namespaces within the `System` namespace. For example, the `System.Collections` namespace contains classes that can be used to implement containers, and the `System.Data` namespace contains the classes needed to work with databases.

Next we have the following function header:

```
int _tmain()
```

Instead of `main()`, the application wizard creates a `_tmain()` function. If you prefer, you can use `main` instead of `_tmain`. Although we won't go into a detailed explanation here, the reason the application wizard automatically uses `_tmain` is because it provides some additional support for Unicode characters.

Inside the `_tmain` function we see the following statement:

```
Console::WriteLine(S"Hello World");
```

This shows a class from the .NET Framework class library being used. This statement calls the `WriteLine` method, which is a member of the `Console` class. The `Console` class is in the `System` namespace. If we hadn't specified that we were using the `System` namespace, we would have to write this statement as:

```
System::Console::WriteLine(S"Hello World");
```

Like `cout`, the `Console::WriteLine` method displays console output.

Program G-2 shows another example. This program uses the code previously discussed to dynamically allocate a managed array of integers. The contents of the array are then displayed using the `Console::WriteLine` method within a loop.

Program G-2

```
1  // This program uses managed C++ code.
2  #include "stdafx.h"
3  #using <mscorlib.dll>
4  using namespace System;
5
6  int _tmain()
7  {
8     // Create an array of integers.
9     const int SIZE = 12;
10    int numbers __gc[] = new int __gc[SIZE];
11
12    // Store some values in the array.
13    for (int i = 0; i < SIZE; i++)
14       numbers[i] = i;
15
16    // Display the values in the array.
17    for (int i = 0; i < SIZE; i++)
18       Console::WriteLine(numbers[i]);
19
20    return 0;
21 }
```

Program Output

```
0
1
2
3
4
5
6
7
8
9
10
11
```

Learning More

An easy way that you can learn more about .NET programming and managed C++ is through the Visual Studio .NET online help. Just click Help on the main menu bar, and then click Contents. You'll find an abundance of information there.