In this chapter you will learn:
♦ About the .NET Framework
♦ How to create managed C++ applications
♦ Memory management techniques with managed extensions
♦ About managed extensions and inheritance
♦ How to create properties
♦ How to create Web services

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years.

John Von Neumann (ca. 1949)

PREVIEW: THE CLEANERS PROGRAM

In this chapter, you will learn to work with Managed Extensions for C++, which allow you to create programs that will function with the .NET Framework. The .NET Framework is capable of executing various types of applications, including console applications. In order to study this chapter’s Managed Extensions for C++ techniques, you will create a simple console application that may be used by a laundry and dry-cleaning service to calculate the cost of a cleaning bill. The program allows you to select either laundry or dry-cleaning services, and prompts you to enter the number of items you need to have cleaned. Then, the program allows you to choose the specific type of garment you need to have cleaned, and presents you with the cost of cleaning the garment along with the total cost of cleaning all of your garments.
To preview the Cleaners program:


2. Copy the Chapter14_Cleaners folder from the Chapter.14 folder on your Data Disk to the Chapter.14 folder in your Visual C++ Projects folder. Then open the Cleaners project in Visual C++.

3. The Cleaners program contains three classes: a GarmentCleaning base class and two classes that derive from it, Laundry and DryCleaning. Open the GarmentCleaning.h file in the Code Editor window. The class uses a new type of statement, a #using statement, to import a file named mscorlib.dll, which contains most of the core classes in the .NET Framework class library. Notice that the class includes several new keywords that begin with double underscores (__). These new keywords are referred to as managed extensions because they are extensions to the C++ language that allow you to write code that is managed by the .NET Framework. The __abstract keyword is used to create an abstract base class, and the __gc keyword is used to create a class that is capable of running on the .NET Framework. The __properties keyword is used with accessor functions to create special data members called properties that conform to the information-hiding techniques discussed earlier in the book. Also notice that the class uses new data types, including String* for string variables and Double (with an uppercase D) for double variables. The GarmentCleaning.h file is shown in Figure 14-1.

4. Now open the main.cpp file in the Code Editor window. One of the main things you should notice about the file is that it contains numerous Console::Write(), Console::WriteLine(), and Console::ReadLine() methods. These methods are the .NET Framework equivalent of the cout and cin

```cpp
#pragma once
#using <mscorlib.dll>
using namespace System;
__abstract __gc class GarmentCleaning
{
public:
    GarmentCleaning(void);
    ~GarmentCleaning(void);
    void setFirstName(String*);
    void setLastName(String*);
    String* getFirstName(void);
    String* getLastName(void);
    void garmentListMenu();
    __property void set_OrderTotal(Double);
    __property Double get_OrderTotal(void);
protected:
    String* pFirstName;
    String* pLastName;
    String* arGarments[];
    Double dTotal;
};
```

Figure 14-1 GarmentCleaning.h
statements you used early on in this book to write information to the standard output stream and read information from the standard input stream. If you scroll through the file, you will see other new programming elements, including the __pin keyword, which prevents an object pointer from being moved around in memory by the .NET Framework. Portions of the main.h file are shown in Figure 14-2.

![Figure 14-2 main.cpp](image)

5. Build and execute the Cleaners program. Figure 14-3 shows the output from a completed version of the program after a user has entered some information.

Be sure to enter only integers when the program prompts you for numeric values. The basic functionality of the program is fairly simple, in order to allow you to concentrate on the Managed Extensions for C++ techniques presented in this chapter. As with several other programs you created early on in this book, the Cleaners program does not include any error checking or ways of validating user input. Therefore, you will find that it is easy to crash the program.

![Figure 14-3 Output of Cleaners program](image)

6. Press any key to close the console window.
INTRODUCTION

All of the C++ programs you have created so far, whether console or MFC applications, have been designed to run within the confines of a single platform. Even though the console applications you have written can be used on almost any platform for which there is a C++ compiler, the programs will still only run within the confines of that platform. MFC applications are even more restrictive, in that they will operate only on a Windows platform. In this chapter, you will learn how to use Managed Extensions for C++ to create code that can be accessed from applications and programming languages that conform to the .NET Framework.

The .NET Framework is a platform designed for creating, deploying, and running Web applications and services. A Web application is a program that executes on a server but that clients access through an HTML document loaded in a Web browser. Some simple types of Web applications can execute entirely on the client's computer. However, most professional Web applications use the Web browser only for displaying information; the actual execution of heavy-duty code and calculations usually takes place on a server. You cannot use C++ to create the HTML portion of a Web application that appears in a Web browser. Instead, you use C++ to write any heavy-duty code and calculations that execute on the server. You then use other .NET Framework programs, such as ASP .NET, to call and execute the C++ server code and display any results in an HTML document loaded in a Web browser.

This chapter assumes a basic understanding of how to work with Web browsers, Web servers, and Internet technology.

A Web service, or XML Web service, is a software component that resides on a Web server. Web services do not contain any sort of graphical user interface or even a command-line interface. Instead, they simply provide services and data in the form of methods and properties; it is up to the client accessing a Web service to provide an implementation for a program that calls a Web service. You can think of a Web service as being just the server-side portion of a Web application. One of the great benefits of Web services is that clients access a Web service's methods and properties using industry standard Web protocols such as HTTP, XML, and SOAP. Do not be concerned if you are not familiar with these Web protocols. Simply understand that they are standard technologies that make it much easier to access a Web service across the Internet.

As an example of a Web service, consider a Web page that displays the prices of commodities that you may track, such as crude oil, natural gas, gold, or silver. The Web page may periodically call methods of a Web service that return the most recent trading price for each type of commodity. The developer of the Web page only needs to know which method of the Web service to call for each type of commodity (such as a getSilverPrice() method that returns the current price of silver). The Web service itself does not care
what you do with the data once you receive it; it is up to you to display it on your Web page, store it in a database, or use it in some other way in your application.

Because C++ enables programmers to write extremely efficient code through its object-oriented programming capabilities and memory management techniques, it is a popular language for writing the server portion of a Web application or for writing Web services. Even though the primary role of C++ in the context of the .NET Framework is to provide code that executes on a server as part of a Web application or Web service, you can also use it to write other types of .NET Framework programs that do not execute on the Web. In this chapter, you will first create the Cleaners program as a .NET Framework console application, so that you can focus on the techniques necessary for writing C++ code that operates within the .NET Framework. Then, you will create a Web service version of the Cleaners program that can be accessed from a Web page.

**The .NET Framework**

In the past, programmers faced many challenges when developing distributed applications for the Internet. The term **distributed application** is used to describe multiple computers sharing the computing responsibility for a single application. One of the biggest challenges programmers face is how components (objects) written in different languages should communicate with each other. One solution that Microsoft came up with is ActiveX. Recall from Chapter 10 that ActiveX is a technology that allows programming objects to be easily reused with any programming language that supports the Microsoft Component Object Model. The Component Object Model, or COM, is an architecture for cross-platform development of client/server applications. ActiveX controls are objects that are placed in Web pages or inside programs created in COM-enabled programming languages. Two cross-platform development architectures that compete with COM are Common Object Request Broker Architecture, or CORBA, developed by an industry consortium known as the Object Management Group (OMG), and Remote Method Invocation, or RMI, developed by Sun Microsystems.

Cross-platform development architectures such as COM, CORBA, and RMI address part of the challenge by allowing components to communicate with each other and by providing common functionality for writing a component, regardless of the language the component was written in. However, although cross-platform development architectures provide a consistent platform for developing distributed applications, they do not provide a consistent platform for deploying and executing them. In order to provide a consistent platform for the development, deployment, and execution of distributed applications, Microsoft developed the .NET Framework. The .NET Framework is not a platform in the traditional sense, in that it is not tied to one particular piece of hardware, as are platforms such as Windows and UNIX. Instead, a .NET Framework application can be executed on any hardware device for which there exists a run-time host. A **run-time host** is a software component capable of loading and executing a specific type of .NET Framework application. For example, Internet Explorer is a run-time host that supports
the downloading and execution of managed controls, which are browser-based controls that are managed by the .NET Framework.

Although the .NET Framework is primarily designed as a platform for creating and running Web applications and services, you can also use it to create other types of solutions, including console and Windows applications. You can use the .NET Framework to create the following types of applications and services:

- ASP .NET applications
- Console applications
- Scripted applications
- Web services
- Windows GUI applications
- Windows services

At the time of this writing, the .NET Framework development environment is available only for Windows operating systems, although implementations for other platforms are in development. However, the .NET Framework includes built-in run-time hosts for each of the preceding application types. In this chapter, you will create .NET Framework console applications and a Web service.

You may be surprised that you will not be creating a Windows GUI application in this chapter. So far in this book, you have created Windows GUI applications using MFCs. However, with the .NET Framework, you create Windows GUI applications using Windows Form controls, which replace ActiveX controls and MFCs in the .NET Framework. In Visual Basic and C#, you can add Windows Form controls using a visual editor, similar to the Dialog Editor you used to create MFC programs in Visual C++. However, Visual C++ does not include a visual editor for adding Windows Form controls. You can add Windows Form controls to a Visual C++ program, but you must add the code manually, similarly to the way you added code for the Windows API calculator you created in Chapter 9. Microsoft may eventually add a visual editor for Windows Form controls to Visual C++, but until they do, you should continue using MFCs to create Windows GUI applications with Visual C++. It is much easier to add the controls using a visual editor than by writing the code manually. Additionally, MFCs are still extremely powerful, and developers will continue to use them to create Windows applications for some time to come. However, if you start working with the C# or Visual Basic programming languages, then you should consider switching to Windows Forms.

In the rest of this section, you will study the following major pieces of the .NET Framework:

- The Common Language Runtime
- Assemblies
The .NET Framework

- The Common Type System
- The Common Language Specification

The Common Language Runtime

Code that you write for the .NET Framework, regardless of the programming language, is called managed code. Code that is not written for the .NET Framework is called unmanaged code. Similarly, entire applications written for the .NET Framework are called managed applications, and applications that are not written for the .NET Framework are called unmanaged applications. Managed code executes in the Common Language Runtime, or CLR, which is the run-time platform for the .NET Framework. The Common Language Runtime is what is actually started and managed by a run-time host. Once a run-time host starts the Common Language Runtime, the Common Language Runtime can then execute any .NET application that was written for it. You can think of the Common Language Runtime as being the .NET Framework’s “operating system,” but without any required hardware. The Common Language Runtime handles all code execution and other requirements that are normally handled by an operating system. When you write and compile code for a particular platform, the platform itself is referred to as the target. In the case of managed code written and compiled for the .NET Framework, the target is the Common Language Runtime.

As you know, you must compile unmanaged C++ code for Windows operating systems in order to be able to execute it. Similarly, you must compile managed code for the Common Language Runtime in order to be able to execute it in the .NET Framework. Code compiled for the Common Language Runtime can then interact with code written in other languages and take advantage of the other benefits of the Common Language Runtime, such as being able to execute wherever a run-time host exists.

Note

The Common Language Runtime replaces COM in the .NET Framework.

When you compile managed code for the Common Language Runtime, the code is first compiled into a special assembly language called Microsoft Intermediate Language, or MSIL. The Common Language Runtime uses a technology called Just-In-Time, or JIT, compilation in order to correctly compile the MSIL code into the native machine code for the hardware type where the run-time host is executing. This process is illustrated in Figure 14-4.
If you are familiar with Java programming, then you will recognize that the Common Language Runtime is similar to the Java Virtual Machine (Java VM), which is the language interpreter for the Java programming language. There is a different Java VM for each platform supported by the Java programming language. To execute a Java program, users need only a copy of the Java VM for their particular platform. Microsoft took the Java VM one step further by not only creating run-time hosts for different platforms (similar to the Java VM), but also allowing programmers to write code for the .NET Framework using any programming language they prefer, including Java.

Although programming language interoperability and the ability to execute wherever a run-time host exists are very important, they are not the only beneficial aspects of the Common Language Runtime. The Common Language Runtime also manages security, performs type checking, manages memory, and performs other important tasks. Although advanced topics such as thread execution and security will not be covered, later in this chapter you will learn about memory management in the Common Language Runtime as it relates to C++ programming.

When you compile .NET Framework code, the compiler creates a portable executable file that contains MSIL code along with metadata, which will be discussed shortly. **Portable executable (PE) file** is the file format for executable programs or for files that will be assembled into executable programs. PE files end with an extension of .exe, .dll, .obj, or .lib. The PE files you work with in this chapter will be executable files with an extension of .exe. Although you will rarely need to work directly with the MSIL code stored in a PE, once your programming skills are more advanced, there will be times when you will want to view it for debugging purposes, or if you are trying to increase your program’s performance. The .NET Framework includes a tool called the MSIL Disassembler that you can use to examine the contents of a PE.

Next, you will create a simple .NET Framework console application using Managed Extensions for C++. The program will print *Hello World* to the screen. For now, do not
The .NET Framework 9

worry about exactly what the term “Managed Extensions for C++” means. Simply understand that Managed Extensions for C++ allow you to use C++ to write applications that target the .NET Framework.

To create a simple .NET Framework console application using Managed Extensions for C++:

1. Create a new project named HelloWorld using the Managed C++ Application Wizard. Save the project in the Chapter.14 folder in your Visual C++ Projects folder. Unlike most Visual C++ wizards, the wizards for Managed C++ applications do not display dialog boxes that allow you to select various options for your project. Instead, the wizard simply creates any necessary files and opens the new project in the IDE.

2. After the Managed C++ Application Wizard creates your project, you should be able to see the files that were created in Solution Explorer. The wizard created a C++ file named HelloWorld.cpp along with stdafx.h and stdafx.cpp files. You will also see a new file that you have not encountered previously, the AssemblyInfo.cpp file. The AssemblyInfo.cpp file will be discussed in the next section.

3. Now, open the HelloWorld.cpp file in the Code Editor window. The file contains some simple code that prints Hello World to the screen. Some portions of the code, such as the Console::WriteLine(S"Hello World"); statement are modified to conform to the .NET Framework. You will study the requirements for writing C++ for the .NET Framework shortly. Figure 14-5 shows an example of the HelloWorld.cpp file; the wizard-generated comments have been removed for brevity.

```c++
#include "stdafx.h"
#include <mscorlib.dll>
#include <tchar.h>
using namespace System;

int _tmain(void) {
    Console::WriteLine(S"Hello World");
    return 0;
}
```

Figure 14-5  Managed C++ console application that prints Hello World to the screen

When you use the Managed C++ Application Wizard, the project is created with a special main() function that reads _tmain(). This is a special Microsoft extension known as a generic-text mapping routine that simplifies code development for international markets. Unless you plan on writing internationalized code, you should continue using the standard main() function.

4. Build and execute the managed HelloWorld project. Figure 14-6 shows the output. You will notice that the program looks and functions exactly the same
as the other types of console applications you have created. The important thing to understand is that the code is actually being executed by the Common Language Runtime in a built-in console application run-time host.

5. Press any key to close the console window.

Now you will use the MSIL Disassembler tool to view the MSIL code contained in HelloWorld.exe. The MSIL code is the PE created for the project after you compile it.

To use the MSIL Disassembler tool to view the MSIL code contained in the HelloWorld.exe:

1. Use Windows Explorer or My Computer to locate the ildasm.exe file, which is the executable file for the MSIL Disassembler tool. By default, the file is located in the C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin folder, although it may be stored in some other location on your system.

2. Depending on how your desktop is configured, click or double-click the ildasm.exe file to open the MSIL Disassembler program window. Figure 14-7 shows how the MSIL Disassembler program window appears when it first opens.

3. Select Open from the File menu to display the Open dialog box. Navigate to the Debug folder beneath the HelloWorld folder. (The HelloWorld folder should be located in the Chapter.14 folder in your Visual C++ Projects folder.) Open the...
The .NET Framework 11

*HelloWorld.exe* file, which is the PE file for the managed HelloWorld program. The MSIL Disassembler program window shows a breakdown of the parts of the HelloWorld.exe PE file, as shown in Figure 14-8.

4. The last item in the window, which begins “main: int32...”, represents the MSIL code for the main() function contained in the HelloWorld.cpp file. Double-click this item, which opens a separate window containing MSIL code, as shown in Figure 14-9. Do not worry about what the code means or how it is structured. Remember that you will rarely, if ever, need to work directly with MSIL code. Simply understand that MSIL code will be translated into machine code when the program executes in a run-time host. In the case of a managed C++ console application, Visual Studio includes a built-in run-time host that is capable of executing .NET Framework console applications on Windows platforms.

5. Close the window displaying the MSIL Code, and then close the MSIL Disassembler program window.
Assemblies

In standard programming environments, in contrast to the .NET Framework, executable and DLL files store only the actual code that gives a program its functionality. Any additional files that are required by the application, such as resource files, are stored separately from the executable or DLL file. Traditional Win32 applications also use a special Windows database known as the System Registry to store Windows system information along with initialization and configuration information for individual applications. This means that information about an application’s version or how to install it is stored separately from the application code and any other files the application needs. Although this organization works well, it is not without problems. The System Registry is a complex database that is accessible by any Windows application. Conflicts between software programs can and do occur. You have probably experienced the problem of installing a new software package, only to find that another program on your system no longer works.

When you deploy a distributed application, the files required by the application, along with configuration and deployment information, need to be as self-contained as possible. In the world of the Internet, there is no central System Registry where programs can store and access necessary information. In order to solve these and other problems, the .NET Framework uses assemblies as the basic unit of construction, deployment, versioning, and security. Every .NET Framework application you develop is an assembly. If your application is composed of multiple software components, then each individual component is stored in its own assembly. When you work with a single component application, such as the Hello World project you created earlier, all assembly information is stored in the PE file that is generated when you compile the application.

An assembly can be composed of multiple files, not just a single PE file.

An assembly can be composed of the following four elements:

- Assembly metadata
- Type metadata
- MSIL code
- Resources such as bitmaps and cursors

**Metadata** is information that describes individual elements managed by the Common Language Runtime. It is through the use of metadata that the .NET Framework applications store configuration and deployment information with an application’s PE file. You attach metadata to code using **attributes**, which are code elements used for annotating other elements in your code, such as classes and methods. You can almost think of attributes and the metadata information they store as a special type of comment that is
The .NET Framework 13

accessible at run time. Attributes and metadata are fairly complex subjects. There are several different types of attributes, and you can also create your own custom attributes.

The only required element in an assembly is the assembly metadata, which is stored in the assembly manifest. The assembly manifest describes how all of the elements in an assembly are structured, along with security and version information.

When you create a managed application using a wizard such as the Managed C++ Application Wizard, Visual Studio automatically creates an assembly for you. After you created the Hello World project, you may have noticed that the Solution Explorer window included a file named AssemblyInfo.cpp. This file allows you to change various predefined attributes that store general information about the assembly manifest, such as application title, description, company name, product name, and version number. Assembly attributes are enclosed within brackets and always begin with assembly: to designate that the attribute applies to an assembly. A pair of parentheses containing a pair of quotation marks follows the attribute name itself. For instance, the statement for the AssemblyCompanyAttribute attribute appears as follows:

```
[assembly:AssemblyCompanyAttribute("")];
```

You add metadata to an attribute by placing it within the quotation marks. For example, to add Course Technology as the metadata for the AssemblyCompanyAttribute attribute, you modify the statement as follows:

```
[assembly:AssemblyCompanyAttribute("Course Technology")];
```

At run time, a process called reflection allows you to access the metadata stored in an attribute. Reflection is a little too advanced for our purposes, so it will not be discussed further. You can find plenty of information on the subject in the MSDN Library.

The Common Type System

Recall that in C++ you must specify a variable’s data type when you declare the variable. Types include primitive data types such as integers and floating-point numbers, advanced types such as arrays and pointers, and user-defined types such as classes. Different programming languages have different ways of working with types. These differences present a serious problem for a platform such as the .NET Framework, which allows programmers to use whatever language they desire to target the Common Language Runtime. In order to provide cross-language compatibility, the Common Language Runtime uses the Common Type System, or CTS, to define how types are declared, used, and managed. The CTS has been designed to function with more than 15 programming languages, ranging from C-based languages such as C++ and Java to Rapid Application Development (RAD) languages such as Visual Basic.

The CTS defines its own versions of primitive data types called built-in value types. Figure 14-10 lists the CTS built-in value types along with their corresponding C++ primitive data types.
CLS stands for Common Language Specification, and it will be discussed in the next section.

Each C++ primitive type is completely interchangeable with its CTS built-in value type. As a consequence, you are not required to use the CTS built-in value types to write code that is .NET Framework compatible. Instead, you can continue using the data types you are familiar with and let the Common Language Runtime handle mapping the C++ data types to the CTS built-in value types. However, you can use the CTS built-in value types if you like. For instance, consider the following C++ primitive declarations:

```cpp
char cLetter = 'A';
short sNumber = 10000;
int iNumber = 100;
float fNumber = 3.5;
double dNumber = 66.6;
bool bValue = true;
```

<table>
<thead>
<tr>
<th>CTS Built-In Value Type</th>
<th>Description</th>
<th>C++ Data Type</th>
<th>CLS-compliant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>An 8-bit unsigned integer</td>
<td>char</td>
<td>Yes</td>
</tr>
<tr>
<td>SByte</td>
<td>An 8-bit signed integer</td>
<td>signed char</td>
<td>No</td>
</tr>
<tr>
<td>Int16</td>
<td>A 16-bit signed integer</td>
<td>short</td>
<td>Yes</td>
</tr>
<tr>
<td>Int32</td>
<td>A 32-bit signed integer</td>
<td>int or long</td>
<td>Yes</td>
</tr>
<tr>
<td>Int64</td>
<td>A 64-bit signed integer</td>
<td>__int64</td>
<td>Yes</td>
</tr>
<tr>
<td>UInt16</td>
<td>A 16-bit unsigned integer</td>
<td>unsigned short</td>
<td>No</td>
</tr>
<tr>
<td>UInt32</td>
<td>A 32-bit unsigned integer</td>
<td>unsigned int or unsigned long</td>
<td>No</td>
</tr>
<tr>
<td>UInt64</td>
<td>A 64-bit unsigned integer</td>
<td>unsigned __int64</td>
<td>No</td>
</tr>
<tr>
<td>Single</td>
<td>A 32-bit single-precision floating-point number</td>
<td>Float</td>
<td>Yes</td>
</tr>
<tr>
<td>Double</td>
<td>A 64-bit double-precision floating-point number</td>
<td>Double</td>
<td>Yes</td>
</tr>
<tr>
<td>Boolean</td>
<td>A Boolean value</td>
<td>Bool</td>
<td>Yes</td>
</tr>
<tr>
<td>Char</td>
<td>A Unicode (16-bit) character</td>
<td>wchar_t</td>
<td>Yes</td>
</tr>
<tr>
<td>Decimal</td>
<td>A 96-bit decimal value</td>
<td>Decimal</td>
<td>Yes</td>
</tr>
<tr>
<td>IntPtr</td>
<td>A signed integer whose size depends on the</td>
<td>No built-in type</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>underlying platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UIntPtr</td>
<td>An unsigned integer whose size depends on the</td>
<td>No built-in type</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>underlying platform</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14-10  CTS and C++ Primitive Data Types
You can write the preceding declarations using CTS built-in value type equivalents as follows:

```csharp
Char cLetter = 'A';
Int16 sNumber = 10000;
Int32 iNumber = 100;
Single fNumber = 3.5;
Double dNumber = 66.6;
Boolean bValue = true;
```

As with C++, the syntax for the CTS built-in value types is case-sensitive. If you use `Double` (with an uppercase `D`) to declare a double data type variable, then you will create a CTS built-in value type variable, because the C++ equivalent (`double`) uses a lowercase `d`.

Although the two type systems are completely interchangeable, it is good practice to use the type system that is the target of your code. Use C++ primitive data types for unmanaged C++ code, and use the CTS built-in value types for managed code.

### The Common Language Specification

In order for .NET Framework code to be truly cross-language compatible, it must conform to a common set of specifications. For instance, C++ uses pointers, which are not found in other languages such as Java and Visual Basic. You would probably cause an error if you tried to pass a C++ pointer to Java, because the Java language would have no idea what to do with the pointer once it received it. In order to ensure that components written for the .NET Framework are compatible with one another, they must conform to the Common Language Specification. The **Common Language Specification**, or **CLS**, defines a set of basic language rules that allows managed code to be cross-language compatible.

It is important to understand that managed code that is not CLS-compliant will still execute in the Common Language Runtime. The only difference is that code that is not CLS-compliant may not be compatible with CLS-compliant managed code written in other languages.

At a beginner level, you probably do not need to know all of the intricate details of the CLS. However, you should understand that the Managed Extensions for C++ techniques discussed later in this chapter enable you to write C++ code that is CLS-compliant. In Figure 14-10, you may have noticed that some of the built-in value types are not CLS-compliant. For instance, unsigned integers, such as UInt32, are not supported in several languages and therefore are not CLS-compliant. However, unsigned integers are commonly used in C++ and are included in the CTS to allow programming languages that support unsigned integers to use them in a .NET Framework application. If you want your code to be completely CLS-compliant, however, you must not use these value types in your .NET Framework applications.
The CTS provides alternatives for all value types that are not CLS-compliant. For instance, the CLS-compliant alternative to the UInt32 value type (which is not CLS-compliant) is Int64. See the entry for each noncompliant CLS value type in the MSDN Library for a CLS-compliant alternative.

Creating Managed C++ Applications

At this point, you should have a pretty good idea of what the .NET Framework is. What may not be clear to you is how C++ fits into the picture. You already know that you cannot easily create managed Windows applications for the .NET Framework. You also know that you can create managed console applications that function in virtually an identical manner as unmanaged console applications. Given these constraints, why write C++ programs for the .NET Framework at all? Why not continue writing unmanaged C++ programs, using the same tools you have used throughout this book? The truth is, if you do not need to write applications that target the .NET Framework, you should continue writing unmanaged C++ code.

However, C++ has a very important place in the .NET Framework, which may not be completely clear to you as a beginning programmer. Without a doubt, C++ is the most powerful programming tool in the Visual Studio suite because of its powerful object-oriented features, ability to compile directly to machine code, and ability to manage memory through pointers and references. Although memory management with pointers and references may seem tedious, it nonetheless gives C++ much greater power than languages such as Visual Basic, which have no similar memory management tools. Because of its power—and extreme popularity—C++ is ideally suited to write server-based applications that can be accessed by .NET Framework applications written in languages such as C# and ASP .NET. For instance, you may have a managed Web application that allows clients to purchase items from an online retailer. You would use ASP .NET to display and manage a “shopping cart” Web page. Then, the ASP .NET component of the application would call a managed C++ component on the server to handle complex processing, such as calculating sales tax and verifying credit card information.

To create managed C++ applications, you use the .NET Framework class library and Managed Extensions for C++. First you will learn about the .NET Framework class library.

The .NET Framework Class Library

The .NET Framework class library is a large collection of classes used for writing programs that conform to the .NET Framework. Classes within this library are organized into namespaces in order to prevent naming conflicts between classes and with any managed code that accesses the library. Figure 14-11 lists some common namespaces in the .NET Framework class library.
The .NET Framework class library contains almost 100 namespaces, with literally thousands of classes stored within the namespaces. (To give you an idea of how large and well developed the .NET Framework class library is, consider that the MFC class library consists of a little more than 200 classes.) You can use the .NET Framework class library to add many different types of functionality to your managed code, ranging from the System.IO namespace, which is used for reading data from and writing data to streams and files, to the System.Web interface, which allows .NET Framework server programs to communicate with a Web browser. Each individual namespace stores one or more classes that perform related tasks. For instance, the System.Security namespace stores classes that provide security functionality for the Common Language Runtime.

As you know, you use the using directive in unmanaged C++ console applications to give your program access to a namespace. For instance, the statement `using namespace std;` gives a C++ program access to the std namespace, which contains all of the names that are defined in the Standard C++ Library. Managed C++ programming introduces the new `#using preprocessor directive`, which allows managed C++ applications to import class

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Contains numerous base classes including classes that define data types, events and event handlers, and other basic programmatic components; Other system classes perform math operations, convert data types, manage the application environment, control local program execution, and control managed and unmanaged applications</td>
</tr>
<tr>
<td>System.Data</td>
<td>Contains classes that allow .NET Framework applications to access data sources</td>
</tr>
<tr>
<td>System.Drawing</td>
<td>Contains classes that provide .NET Framework applications with basic graphics functionality</td>
</tr>
<tr>
<td>System.IO</td>
<td>Contains classes that allow .NET Framework applications to read data from and write data to streams and files</td>
</tr>
<tr>
<td>System.Messaging</td>
<td>Contains classes that allow .NET Framework applications to include messaging functionality</td>
</tr>
<tr>
<td>System.Net</td>
<td>Contains classes that give .NET Framework applications access to numerous network protocols</td>
</tr>
<tr>
<td>System.Security</td>
<td>Contains classes that provide security functionality for the Common Language Runtime</td>
</tr>
<tr>
<td>System.Threading</td>
<td>Contains classes that enable .NET Framework applications to perform multithreaded processing</td>
</tr>
<tr>
<td>System.Web</td>
<td>Contains classes that allow .NET Framework server applications to communicate with a Web browser</td>
</tr>
<tr>
<td>System.Xml</td>
<td>Contains classes that allow .NET Framework applications to perform XML processing</td>
</tr>
</tbody>
</table>

Figure 14-11  Common namespaces in the .NET Framework class library
libraries compiled for the Common Language Runtime. You use the #using preprocessor directive to import the class libraries in **mscorlib.dll**, which contains most of the core classes in the .NET Framework class library. All managed C++ applications must reference the mscorlib.dll file.

To reference the mscorlib.dll file, you use the same syntax that you use for the #include directive. You place the mscorlib.dll name within angle brackets (<>), and you do not include a semicolon at the end of the statement. After importing the metadata in mscorlib.dll using the #using preprocessor directive, you use the using directive to give your program access to any of the namespaces in the .NET Framework class library. For instance, many of the basic classes in the .NET Framework class library are stored in the System namespace. To give a managed C++ application access to the System namespace, you must include the following two statements in your code:

```cpp
#using <mscorlib.dll>
using namespace System;
```

Next, you will start creating the managed Laundry Service application.

To start creating the managed Laundry Service application:

1. Create a new project named **Cleaners** using the Managed C++ Empty Project Wizard. Save the project in the **Chapter.14** folder in your Visual C++ Projects folder.

2. Select **Add Class** from the Project menu to add a new generic class named **GarmentCleaning** to the project. In the Generic C++ Class Wizard dialog box, select **public** access, do not type the name of a base class, and leave the Virtual destructor and Inline check boxes cleared. The GarmentCleaning.h file opens in the Code Editor window.

3. In the **GarmentCleaning.h** header file, type the two statements shown in Figure 14-12, which give the class access to the System namespace.

![GarmentCleaning.h file modified to access the System namespace](image)
4. Next, add two new classes: **Laundry** and **DryCleaning**. Derive both classes from the GarmentCleaning base class. In the Generic C++ Class Wizard dialog box for both classes, select **public** access, type **GarmentCleaning** as the name of a base class, and leave the Virtual destructor and Inline check boxes cleared.

5. Select **Add New Item** from the Project menu and add a new C++ source file named **main.cpp**. After the file is created add the following code, which includes the Laundry and DryCleaning classes along with an empty main() function.

```cpp
#include "Laundry.h"
#include "DryCleaning.h"
void main() {
}
```

6. Build the project. Although the project does not yet contain any executable code, it should compile normally.

**The Console Class**

The .NET Framework equivalents to the iostream classes you have used frequently throughout this book are located in the Console class, which is located within the System namespace. The Console class contains two methods for writing data to the standard output stream: Write() and WriteLine(). The Write() method prints information to the screen and is essentially equivalent to the cout object you use with unmanaged C++ console applications. The WriteLine() method also prints information to the screen, but followed by a line break. (You can think of the WriteLine() method as a combination of the cout and endl objects.) You pass to each method either a string variable or a literal string. In order for your program to successfully locate the Write() and WriteLine() methods within the System namespace, you must preface both methods in your code with the name of the Console class and the scope resolution operator. Figure 14-13 shows an example of a managed C++ console application that uses both the Write() and WriteLine() methods. Figure 14-14 shows the output.

```cpp
#include <mscorlib.dll>
using namespace System;
void main() {
    Console::Write("Bears ");
    Console::Write("live ");
    Console::WriteLine("in ");
    Console::WriteLine("the ");
    Console::WriteLine("woods. ");
}
```

**Figure 14-13** Managed C++ console application with Write() and WriteLine() methods
The Console class also contains two methods for reading data from the standard input stream: `Read()` and `ReadLine()`. The `Read()` method returns a single character from the standard input stream. If a user types multiple characters, only the first character typed will be returned. The following code uses the `Read()` method to assign a single character to a variable of the CTS built-in value type Char, then prints the variable using the `WriteLine()` method:

```cpp
Console::Write("Enter a character: ");
Char cCharacter = Console::Read();
Console::Write("You typed the character ");
Console::WriteLine(cCharacter);
```

The `ReadLine()` method returns a string from the standard input stream. The following code uses the `ReadLine()` method to assign a string to a variable of the .NET Framework `String` class (you will study the .NET Framework `String` class next), then prints the variable using the `WriteLine()` method:

```cpp
Console::Write("Enter a string: ");
String* pString = Console::ReadLine();
Console::Write("You typed ");
Console::WriteLine(pString);
```

Next, you will use the Console class to add some output statements to the `main()` function in the Cleaners project.

To use the Console class to add some output statements to the `main()` function in the Cleaners project:

1. Return to the `main.cpp` file in the Code Editor window.
2. Add to the `main()` function the output statements shown in Figure 14-15 that appear when the program first executes.
3. Build and execute the program. Figure 14-16 shows the program’s output in the managed console application window.
4. Press any key to close the console window.
The String Class

In this book you have used three different methods to store and manipulate text strings: C-style text strings, the C++ string class, and the MFC CString class. Each method is unique to its original programming language: C-style text strings were originally used in the C language; the string class is part of C++; and the CString class is part of MFCs. Therefore, it should come as no surprise to you that the .NET Framework also has its own method of storing and manipulating strings. In the .NET Framework, you store and manipulate strings through the String class, which is part of the System namespace. Because of the many different ways of storing and manipulating strings (each programming language has its own methods), it makes sense that the .NET Framework would provide a common way to store and manipulate strings that can be used by all programming languages that target the Common Language Runtime. Additionally, providing a common way to store and manipulate strings helps ensure that any string literals used by your application are CLS-compliant.

You declare a variable of the String class as a pointer, using a statement similar to the following:

```cpp
String* pLastName = "Singh";
```

String literals in standard C++ are composed of single-byte strings, which are sequences of ASCII characters in which each character occupies a single byte of memory. However, in order for string literals to be CLS-compliant, they must be in Unicode format. Unicode is a specification used to represent all modern characters, including characters in other languages (such as Japanese), as well as various other types of technical symbols. Unicode string
literals are referred to as **wide-character strings** because each character in the string occupies two bytes. Even though the statement

```
String* pLastName = "Singh";
```

is created using the .NET Framework String class, the string literal assigned to the pLastName variable is a single-byte string. It is, therefore, not CLS-compliant. This means that although the statement will run in the Common Language Runtime, it may not be compatible with managed code written in other languages. You can convert any ASCII string to a wide-character string by adding an uppercase letter L to the beginning of the string, just before the opening double quotation mark. To convert the literal string “Singh” to a wide-character string (and make it CLS-compliant), you use the following statement:

```
String* pLastName = L"Singh";
```

Another method of working with string literals in the .NET Framework is to prefix each literal string with an uppercase S. Applying an S prefix to a literal string creates a single instance of the String class. This single String object instance will then manage all of the string literals in your code. In comparison, the L prefix instantiates a new String object for each literal string you apply it to.

If you are having trouble understanding how this works, recall from Chapter 6 that a string literal is actually a pointer to the memory address of a character array’s first element. The single String object instance actually manages all of the pointers to the memory address of each string literal’s first character. One of the main benefits of the S prefix is better performance, because you do not have to instantiate additional String objects for each of the string literals in your code. Another benefit is that applying the S prefix automatically handles the conversion of the string literal to Unicode, making it CLS-compliant. As an example, the following code uses the S prefix to create a single String object instance for two string literals:

```
String* pLastName = S"Singh";
String* pFirstName = S"Rajeesh";
```

The String class contains various methods for manipulating strings. Figure 14-17 lists the common String class methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone()</td>
<td>Returns a reference to an existing String variable</td>
</tr>
<tr>
<td>Compare()</td>
<td>Compares two strings</td>
</tr>
<tr>
<td>Concat()</td>
<td>Appends one string to another</td>
</tr>
<tr>
<td>Copy()</td>
<td>Replaces the contents of one string with the contents of another</td>
</tr>
<tr>
<td>CopyTo()</td>
<td>Copies specified characters from one string to a specified location in an array of Unicode characters</td>
</tr>
<tr>
<td>EndsWith()</td>
<td>Determines whether a string ends with a set of specified characters</td>
</tr>
<tr>
<td>Equals()</td>
<td>Determines whether two strings contain the same value</td>
</tr>
</tbody>
</table>

*Figure 14-17  Common String class methods*
As with the C++ string class, you can use the assignment operator to assign a new value to a String class variable after declaration, as follows:

```cpp
String* pFirstName = S"Rajeesh";
pFirstName = S"Tasneem";
```

You can also use the assignment operator to copy the contents of one variable to another, as follows:

```cpp
String* pFirstName1 = S"Rajeesh";
String* pFirstName2 = pFirstName1;
```

The preferred method of assigning a new string value to an existing variable is to use the Copy() method. The syntax of the Copy() method is as follows:

```cpp
String::Copy(destination, source);
```

The `destination` argument represents the String variable to which you want to assign a new value. The `source` variable represents the literal string or variable containing the string you want to assign to the destination variable. For example, you would use the following code to assign the value of the `pFirstName1` object to the `pFirstName2` object:

```cpp
String* pFirstName1 = S"Rajeesh";
String* pFirstName2 = String::Copy(pFirstName1);
```

The String class does not include an overloaded addition operator (+) for combining strings, so you need to use the Concat() method, which is similar to the C-style strcat() function. The Concat() method uses the syntax:

```cpp
String::Concat(destination, string1, string2, ...);
```

Notice that you can pass two or more strings to the Concat() method. For instance, the following code uses the Concat() method to build a String variable named `pWholeName` using the `pFirstName` variable, a string literal containing a single space, and the `pLastName` variable:

```cpp
String* pLastName = S"Singh";
String* pFirstName = S"Rajeesh";
String* pWholeName = String::Concat(pFirstName, S" ", pLastName);
```

The preferred method of determining if two String class variables contain the same text uses the String class Equals() method. The Equals() method uses the syntax:

```cpp
String::Equals(string1, string2);
```

and compares the value of `string1` to the value of `string2`. The following example uses the Equals() method in an if statement’s conditional expression:

```cpp
String* pFirstName1 = S"Rajeesh";
String* pFirstName2 = S"Tasneem";
if (String::Equals(pFirstName1, pFirstName2))
    Console::WriteLine("Same name");
else
    Console::WriteLine("Different name");
```
If you write a .NET Framework program that performs a large amount of string manipulation, the String class is actually not your best choice. The String class is considered to be immutable, which means it cannot change. Anytime you call a method such as Copy() or Concat(), the Common Language Runtime creates an entirely new version of the string in memory. For heavy string manipulation, this can exact a high cost on performance. A better choice for string-intensive applications is to use the StringBuilder class, which is contained in the System.Text namespace. The StringBuilder class is considered to be mutable, which means it can change. The StringBuilder class allows you to dynamically manipulate strings. However, the StringBuilder class does not come without a price. It allows you to dynamically manipulate strings by allocating more memory than the strings need. If your application is not particularly string-intensive, the extra memory that the StringBuilder class allocates may be unnecessary.

Next, you will use the String class to add some String* pointer variables and accessor functions to the GarmentCleaning base class. Note that String* pointers can be used only inside managed classes. Although you will add String* pointers to your code in this exercise, you will not be able to build your project until you turn your classes into managed classes, which you will learn how to do later in this chapter.

To use the String class to add variables and accessor functions to the GarmentCleaning base class:

1. Open the GarmentCleaning.h file in the Code Editor window.
2. As shown in Figure 14-18, create a new protected section in the interface file and add two String* data member declarations, which store a customer’s first and last names.

```
protected:
    String* pFirstName;
    String* pLastName;
```

**Figure 14-18** String* data members and accessor functions added to GarmentCleaning.h
3. Also as shown in Figure 14-18, add four accessor functions, which set and retrieve the values in the two new String* data members, to the public section.

4. Open the **GarmentCleaning.cpp** file in the Code Editor window.

5. At the bottom of the file, add the following definitions for the new member functions.

   ```cpp
   void GarmentCleaning::setFirstName(String* pName) {
       pFirstName = pName;
   }
   void GarmentCleaning::setLastName(String* pName) {
       pLastName = pName;
   }
   String* GarmentCleaning::getFirstName(void) {
       return pFirstName;
   }
   String* GarmentCleaning::getLastName(void) {
       return pLastName;
   }
   ```

6. Next, open **main.cpp** in the Code Editor window.

7. As shown in Figure 14-19, add statements that declare two local String* variables and receive the customer’s first and last name through Console::ReadLine() statements. Also add Console::Write() statements that prompt the user for the names and store them in the local variables. You will later add code that assigns the values stored in the local String* variables to the associated String* data members in the GarmentCleaning class.

   ```cpp
   void main() {
       Console::Write("Gosselin Laundry ");
       Console::WriteLine("and Dry Cleaning Services");
       String* pFirstName;
       String* pLastName;
       Console::Write("Enter the customer’s first name: ");
       pFirstName = Console::ReadLine();
       Console::Write("Enter the customer’s last name: ");
       pLastName = Console::ReadLine();
   }
   ```

8. Save the Cleaning project.
You will not be able to build the project until you learn about managed types. Before you study managed types, however, you need to learn about the Convert class.

**The Convert Class**

Because the ReadLine() method returns only strings, you cannot assign its return value to any data type other than the .NET Framework String class. If the data you gather with the ReadLine() method is a numeric value that you need to use in a calculation, you will need to convert the value from the String type to the correct primitive data type. The Convert class of the .NET Framework class library contains methods for converting primitive data types to other primitive data types. For instance, the ToInt32() method converts various data types to the CTS built-in value type of Int32, which is equivalent to the C++ int and long primitive data types. The syntax for using a Convert class conversion method is as follows:

\[
\text{type variable} = \text{Convert::conversion\_method(variable);} \\
\]

The variable name in the left operand is the variable to which you will assign the converted value; it must be of the same data type as the conversion function. The right operand calls the Convert class along with the specific conversion method, passing to it the name of the variable to be converted.

The following code shows an example of how to convert a string received from the ReadLine() method to a double number.

```c++
Console::WriteLine("Enter a double number:");
String* pDoubleString = Console::ReadLine();
double dNewDouble = Convert::ToDouble(pDoubleString);
Console::Write("You entered ");
Console::WriteLine(dNewDouble);
```

See the Convert class in the MSDN Library for a complete listing of conversion methods.

You can use the methods of the Convert class whenever you need to convert primitive data types, not just when you need to convert data received from the standard input stream.

Next, you will use the Convert class to convert a string received from the input stream to an Int32 variable.
To use the Convert class to convert a string received from the input stream to an Int32 variable:

1. Return to the `main.cpp` file in the Code Editor window.

2. As shown in Figure 14-20, add three Console class output statements, which prompt users to select the type of cleaning they need. Add a fourth statement that declares a String* variable named pResponse and assigns to it the value received from the input stream. The fifth new statement declares an Int32 variable named iType and assigns to it the value stored in pResponse after using the ToInt32() method to convert the value to an integer.

3. Also as shown in Figure 14-20, add a statement that prompts the user to enter the number of items that need to be cleaned. The code reuses the pResponse String* variable, but declares a new Int32 variable named iCount that will store the number of items.

4. Save the Cleaning project.

**Managed Extensions for C++**

Although the .NET Framework class library allows you to add many different types of functionality to your managed code, simply using its classes does not automatically turn your C++ code into a managed application. You need to use managed extensions to turn your C++ applications into managed applications. **Managed extensions** are new keywords that have been added to C++ that allow you to modify classes, functions, and other programming elements so that they function within the Common Language Runtime. Figure 14-21 lists the managed extension keywords. All of the keywords are easily identifiable because they start with a double underscore (___).
The rest of this chapter is devoted to explaining managed extension keywords and demonstrating how to use them to create managed C++ applications.

**Memory Management with Managed Extensions**

As you know, to manage memory you manipulate the memory addresses where variables are stored. This gives you much greater control over your program and allows you to adjust memory requirements as necessary, which will increase the performance of your program. However, not all programming languages support memory management techniques. The .NET Framework uses a process called garbage collection to handle memory management for managed applications. The Common Language Runtime garbage collector allocates and deallocates memory on a managed heap for .NET Framework applications. One of the most important tasks in writing managed C++ code is to modify your classes and pointers with managed extension keywords so that memory for them is managed by the garbage collector. If you do not modify your classes and pointers to work with the garbage collector, then they will be considered unmanaged code and will not be recognized by the Common Language Runtime.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__abstract</td>
<td>Declares an abstract class</td>
</tr>
<tr>
<td>__box</td>
<td>Creates on the Common Language Runtime heap a copy of a __value class</td>
</tr>
<tr>
<td>__delegate</td>
<td>Declares a reference to a managed class function pointer</td>
</tr>
<tr>
<td>__event</td>
<td>Declares an event method of a managed class</td>
</tr>
<tr>
<td>__finally</td>
<td>Declares a finally block as part of a try-finally statement</td>
</tr>
<tr>
<td>__gc</td>
<td>Declares a gc type</td>
</tr>
<tr>
<td>__identifier</td>
<td>Enables the use of C++ keywords as identifiers</td>
</tr>
<tr>
<td>__interface</td>
<td>Declares an interface</td>
</tr>
<tr>
<td>__nogc</td>
<td>Declares an unmanaged class</td>
</tr>
<tr>
<td>__pin</td>
<td>Prevents an object or embedded object of a managed class from being moved by the Common Language Runtime during garbage collection</td>
</tr>
<tr>
<td>__property</td>
<td>Declares a property member for a managed class</td>
</tr>
<tr>
<td>__sealed</td>
<td>Prevents a gc class from being used as a base class, or a method from being overridden by a derived class method</td>
</tr>
<tr>
<td>__try_cast</td>
<td>Performs a cast or throws an exception if the cast fails</td>
</tr>
<tr>
<td>__typeof</td>
<td>Returns the corresponding CTS built-in value type of a native C++ primitive type</td>
</tr>
<tr>
<td>__value</td>
<td>Declares a value type</td>
</tr>
</tbody>
</table>

**Figure 14-21** Managed extension keywords
Although this book has consistently used the term *class* throughout, recall from Chapter 5 that classes are also referred to as user-defined data types or programmer-defined data types. In the .NET Framework, the preferred term for a class is simply *type*, so that term will be used frequently for the rest of this chapter. Whenever you see the term class or type, remember that they refer to the same thing.

### Unmanaged and Managed Types

All of the types in a managed application do not necessarily need to be managed types. For instance, you may have some behind-the-scenes unmanaged types that are used to perform complex processing, which a client never needs to see. The only types that need to be managed are the types that must be visible to the Common Language Runtime. By default, any unmanaged types that are included in a managed application are marked with the *__nogc* keyword (for “no garbage collection”). You do not need to explicitly declare an unmanaged type with *__nogc* because it is implied by default. However, it is good practice to explicitly declare any unmanaged types in your program with *__nogc* in order to make it clear that they are not managed. For instance, to explicitly declare a class named Stocks as unmanaged, you add the *__nogc* keyword to the start of the class declaration, as follows:

```cpp
__nogc class Stocks {
  public:
    Stocks();
...
```

There are two basic managed types you can create for managed C++ programs: value and gc. First, the value type will be discussed.

#### Value Types

In addition to a managed heap, the Common Language Runtime also includes a managed stack. **Value types** are classes whose declarations are defined with *__value* keyword. Objects instantiated from a value type are stored on the managed stack instead of the managed heap. You use value types for small classes with short lifetimes that do not justify the overhead involved with garbage collection. Value types derive from the `System.ValueType` class, which in turn derives from the `System.Object` class. To define the Stocks class as a value type, you add the *__value* keyword to the start of the class declaration, as shown in the following Stocks class declaration:

```cpp
__value class Stocks {
  public:
    Stocks();
...
```

Built-in CTS data types such as `Int32` are actually value types that are stored on the managed stack instead of the managed heap.
In order to instantiate an object of the class on the managed stack, you use the same syntax as you would use to create a class object on the unmanaged stack, as follows:

```cpp
Stocks stockPick;
```

You can then use the managed stock object as you normally would, by calling its methods and properties through the member selection operator:

```cpp
stockPick.setName(S"Cisco");
stockPick.setNumShares(100);
stockPick.setPricePerShare(68.875);
```

You need to follow a number of rules when creating value types. They cannot include destructors or copy constructors, and they cannot have base classes. For example, you cannot use value types in the Cleaning program because the Laundry and DryCleaning classes derive from the GarmentCleaning base class.

You can view a complete list of value type rules in the `__value` Classes topic in the MSDN Library.

### gc Types

A gc (for “garbage collected”) type, or reference type, is stored on the managed heap and is managed by the garbage collector. Directly or indirectly, all gc types derive from the System.Object class, which is the base class for all .NET classes. You create classes as gc types when you want to dynamically allocate objects derived from the class on the heap. You create a gc type by adding the `__gc` keyword to a class declaration. For instance, to define the Stocks class as a gc type, you add the `__gc` keyword to the start of the class declaration, as follows:

```cpp
__gc class Stocks {
    public:
        Stocks();
    ...
}
```

As with value types, you need to follow a number of rules when creating gc types. They cannot inherit from an unmanaged class, nor can an unmanaged class be derived from a managed class. Additionally, gc classes cannot declare a user-defined copy constructor, friend classes, or friend functions.

You can view a complete list of value type rules in the `__gc` Classes topic in the MSDN Library.

Next, you will add the `__gc` keyword to the GarmentCleaning, Laundry, and DryCleaning classes in the Cleaning project. The GarmentCleaning, Laundry, and DryCleaning classes
are by no means large enough to justify storing them on the managed heap. However, you will convert them to gc classes for practice.

To add the __gc keyword to the GarmentCleaning, Laundry, and DryCleaning classes in the Cleaning project:

1. Open the GarmentCleaning.h file in the Code Editor window, and modify the class header declaration so it includes the __gc keyword, as follows:

   __gc class GarmentCleaning

2. Open the Laundry.h file in the Code Editor window, and modify the class header declaration so it includes the __gc keyword, as follows:

   __gc class Laundry :
       public GarmentCleaning

3. Finally, open the DryCleaning.h file in the Code Editor window, and modify the class header declaration so it includes the __gc keyword, as follows:

   __gc class DryCleaning :
       public GarmentCleaning

4. Build and execute the program. Figure 14-22 shows the output.

5. Press any key to close the console window.

In order to instantiate a class object on the managed heap, you use the new keyword with the same syntax you use to create a class object on the unmanaged heap. For instance, regardless of whether the Stocks class is a gc type or an unmanaged type, you can instantiate an object from it using the following statement:

    Stocks* pStockPick = new Stocks;

You can use a managed heap object as you would use an unmanaged heap object, by calling its methods and properties through the indirect member selection operator, as follows:

    pStockPick->setName(S"Cisco");
    pStockPick->setNumShares(100);
    pStockPick->setPricePerShare(68.875);
After you are through with the object, however, you are not required to delete it using the `delete` keyword. Instead, the garbage collector handles the object deletion. Nevertheless, you may still want to delete the object yourself in order to execute any cleanup code in the class destructor. However, an important rule you need to remember is that you cannot manually delete a managed object unless the object's class includes a destructor.

Next, you will add code to the Cleaning project that instantiates either a Laundry object or a DryCleaning object on the managed heap, depending on user input. You will also create two unmanaged functions, `LaundryOrder()` and `DryCleaningOrder()`, for the Laundry and DryCleaning objects that will handle calling the methods and properties of each object. Each function will accept four parameters: a pointer for the object type, two `String*` variables (for the client's first and last names), and an `Int32` variable for the number of items.

To add code to the Cleaning project that instantiates either a Laundry object or a DryCleaning object on the managed heap:

1. Open the `main.cpp` file in the Code Editor window.
2. Just above the `main()` function, add the following function prototypes for the `LaundryOrder()` and `DryCleaningOrder()` functions:

   ```cpp
   void LaundryOrder(Laundry*, String*, String*, Int32);
   void DryCleaningOrder(DryCleaning*, String*, String*, Int32);
   ```

3. Add the following code to the end of the `main()` function, just above the closing brace. The `if...else` statement determines whether the user wants to create a Laundry or DryCleaning object. The appropriate unmanaged function for each object type is called and passed the object pointer, along with the `pFirstName`, `pLastName`, and `iCount` variables.

   ```cpp
   if (iType == 1) {
       Laundry* pLaundryOrder = new Laundry;
       LaundryOrder(pLaundryOrder, pFirstName, pLastName, iCount);
   }
   else if (iType == 2) {
       DryCleaning* pDryCleaningOrder = new DryCleaning;
       DryCleaningOrder(pDryCleaningOrder, pFirstName, pLastName, iCount);
   }
   else
       Console::WriteLine("You entered a wrong character.");
   ```

4. Following the `main()` function, add the following two definitions for the `LaundryOrder()` and `DryCleaningOrder()` functions. Each function contains statements that call inherited `setFirstName()` and `setLastName()` statements.
void LaundryOrder(Laundry* pLaundry, String* pFirst, String* pLast, Int32 iCount) {
    pLaundry->setFirstName(pFirst);
    pLaundry->setLastName(pLast);
}

void DryCleaningOrder(DryCleaning* pDryCleaning, String* pFirst, String* pLast, Int32 iCount) {
    pDryCleaning->setFirstName(pFirst);
    pDryCleaning->setLastName(pLast);
}

5. Build and execute the program. It should function the same as it did before adding the code to instantiate the managed heap objects.

6. Press any key to close the console window.

**Managed Pointers**

When you declare an object on either the managed or unmanaged heap, the object’s memory address is stored in a pointer variable. For instance, the following statement declares a pointer variable named pStockPick that stores the memory address of a new Stocks object stored on the managed heap. (The following statement assumes that the Stocks class declaration includes the __gc keyword.)

```c++
Stocks* pStockPick = new Stocks;
pStockPick->setName(S"Cisco");
pStockPick->setNumShares(100);
pStockPick->setPricePerShare(68.875);
```

A traditional pointer that points to a memory address on the C++ unmanaged heap is called an **unmanaged pointer**. A pointer that points to a memory address on the managed heap is called a **managed pointer** or gc pointer. In the preceding example, pStockPick is a managed pointer. By default the __gc keyword is applied to any pointer that points to an object on the managed heap whose class declaration also includes the __gc keyword. You can also explicitly declare a managed pointer by applying the __gc keyword to the pointer declaration, as in the following example:

```c++
Stocks __gc* pStockPick = new Stocks;
```

Applying the __gc keyword in the preceding statement is unnecessary because it is already applied by default. It is important to understand that although you have complete control over unmanaged pointers, the Common Language Runtime has complete control over managed pointers. As necessary, the garbage collector moves objects around on the managed heap for better performance. If all of your code is managed, you will not notice the garbage collector moving objects around on the managed heap, because the Common Language Runtime automatically updates the addresses stored in your code’s managed pointers. However, if you pass the memory address of a managed type to an unmanaged function, the contents of the memory address may unexpectedly...
change as the garbage collector does its work. For instance, you may have a managed Lease object that gathers information for a commercial real estate lease. One of the pieces of information gathered by the managed Lease object may be the number of square feet. Your program may pass the memory address of the managed Lease object to an unmanaged calcCarpetCost() function that performs a cost calculation. Because the calcCarpetCost() function is unmanaged, it will not be able to find the updated address of the Lease object after the garbage collector moves it around in memory.

To address this problem, you can use the __pin keyword to pin a managed pointer, which prevents the object of the pointer from being moved by the Common Language Runtime during garbage collection. To pin a managed pointer, you use the following syntax: class_name __pin* object_name = new class_name;. For instance, to pin a managed pointer named pLeaseInfo that is instantiated from the Lease class, you use the statement Lease __pin* pLeaseInfo = new Lease;. Figure 14-23 shows a simple version of the Lease class program that passes a pinned pointer to the unmanaged calcTotalCost() function. The class header and function definitions are excluded for brevity. Figure 14-24 shows the output.

```c++
// Lease.h
#pragma once
__gc class Lease
{
public:
    Lease(void);
...
// Lease.cpp
#include "stdafx.h"
#include "lease.h"
using namespace System;
...
void calcTotalCost(Lease* pInfo, double dPrice)
{  
double dSquareFeet = pInfo->getSquareFeet() * 1.1;
double dTotalCost = dSquareFeet * dPrice;
    Console::Write("Total square footage increased by 10% ");
    Console::WriteLine("to account for waste material.");
    Console::Write("Your new square footage is: ");
    Console::WriteLine(dSquareFeet);
    Console::WriteLine(dTotalCost);
}

void main(void)
{  
    Lease __pin* pLeaseInfo = new Lease;
    Console::Write("Enter the number of square feet to carpet: ");
    String* pFeet = Console::ReadLine();
pLeaseInfo->setSquareFeet(Convert::ToDouble(pFeet));
    Console::Write("Enter the carpeting cost per square foot: ");
    String* pCost = Console::ReadLine();
double dPricePerFoot = Convert::ToDouble(pCost);
calcTotalCost(pLeaseInfo, dPricePerFoot);
}
```

Figure 14-23  Lease class program with a pinned pointer
Pinning a pointer significantly decreases a managed application’s performance because the Common Language Runtime cannot make optimum use of the managed heap by allowing the garbage collector to move memory addresses as necessary. Therefore, you should use pinned pointers sparingly.

Once you are finished using a pinned pointer, you can unpin it by assigning a value of zero to it. For instance, you would use the following statement to unpin the pLeaseInfo pointer:

```cpp
pLeaseInfo = 0;
```

You can still use a pointer after you unpin it; unpinning simply allows the garbage collector to move the memory address of the pointer around in memory whenever necessary.

You can also pin individual members of a managed class. However, doing so will actually pin the entire class object.

In the last exercise, you created two unmanaged functions that each receives a managed pointer. In order to ensure that the correct memory addresses of the managed objects are available to the unmanaged functions, you will pin each managed heap pointer.

To pin the managed heap pointers in the Cleaners program:

1. Return to the `main.cpp` file in the Code Editor window.
2. At the end of the `main()` function, modify the statements in the `if...else` structure that instantiate each managed object so that they include the `__pin` keyword, as shown in Figure 14-25.
3. Build and execute the program. It should function the same as it did before pinning the managed heap objects.
4. Press any key to close the console window.
Recall that all gc types derive either directly or indirectly from the System.Object class. This means that all gc types are also Object* pointers, because they are pointers to objects derived indirectly from the Object class. This is important to understand because many of the methods in the .NET Framework class library, along with methods you may create yourself, accept an argument of Object*, meaning that you can pass a gc type to these methods. There may be times when you need to pass a value type to a function argument that calls for an Object* pointer. Value types also derive indirectly from the Object class. However, because they are stored on the managed stack instead of the managed heap, you cannot pass them as a reference to an Object* argument as you could a gc type.

You can use the __box keyword to create a managed copy of a value type through a process called boxing. After you box a value type, you can use it wherever you would use a gc type. Boxing is an advanced topic, so it will not be discussed further here. You can find more information on boxing, and the related topic of unboxing (which removes the “box” from a boxed value type), in the MSDN Library.

**Managed Arrays**

In a managed application, if you create a dynamic array with a C++ primitive data type, then the array will be an unmanaged array. However, in a managed application, if you create a dynamic array with CTS built-in value types, the array becomes a managed array. A managed, or gc, array is stored on the managed heap and is controlled by the garbage collector. For instance, the following code declares a dynamic array that is unmanaged because it is declared using the C++ int data type. The code also deletes the array name when it is through with it.

```cpp
int* arInvestments = new int[10];
...
delete[] arInvestments;
```
In comparison, a dynamic array declared using the built-in CTS Int32 value type is automatically a managed array. When you declare a managed array, you place an additional set of empty brackets following the array name, using the syntax `type array_name[] = new type[elements]`. As with other dynamically managed types, you do not need to use the delete keyword to manually delete a managed dynamic array. The following statement declares a managed array named `arInvestments` of the Int32 data type with ten elements.

```csharp
Int32 arInvestments[] = new Int32[10];
```

Notice in the preceding statement that the dynamic array name is not created as a pointer. Managed dynamic arrays declared using a CTS built-in value type are created as objects, not pointers. This means that you should not include an asterisk after a CTS built-in value type, as you would with unmanaged dynamic arrays of C++ primitive data types. Keep in mind that this rule only applies to arrays of CTS built-in value types. Arrays of gc class objects must be created as pointers. For instance, objects of the String class are gc types. For this reason, you must include the asterisk following the name of the String class (or any type of class) in a managed array declaration using a statement such as `String* arDepartments[] = new String*[];`.

Because managed arrays are created as objects, you cannot use the array name to perform pointer arithmetic. However, you can still perform pointer arithmetic with managed arrays by creating a managed pointer that points to the array name.

Next, you will declare a managed dynamic array named `arGarments` in the GarmentCleaning base class. The Laundry and DryCleaning derived classes will use the `arGarments` managed dynamic array to store a description of garment types. You are creating the array as a dynamic array because the total number of types of garments is different in each derived class. Therefore, each derived class must set the number of elements that the array will store.

To declare and initialize a dynamic array:

1. Open the `GarmentCleaning.h` file in the Code Editor window.
2. Add a declaration in the protected section for a `String*` array named `arGarments[]` using the statement `String* arGarments[];`.
3. Next, open the `Laundry.cpp` file in the Code Editor window.
4. Add to the constructor function the statements shown in Figure 14-26, which initialize the `arGarments[]` array to four elements and assign values to each element.
5. Now open the **DryCleaning.cpp** file in the Code Editor window.

6. Add to the constructor function the statements shown in Figure 14-27, which initialize the `arGarments[]` array to ten elements and assign values to each element.

There may be cases when you want to continue using C++ primitive data types in managed code. For instance, you may want to continue using C++ primitive data types to maintain backward-compatibility with unmanaged C++ code. However, even though your code may use C++ primitive data types, you may still want to take advantage of the benefits of managed arrays. You can create a managed dynamic array of C++ primitive data types by using the `__gc` keyword in the array declaration, using the syntax...
type array_name __gc[] = new type __gc[elements];. For example, the following statement declares a managed dynamic array of C++ int data types:

    int arInvestments __gc[] = new int __gc[10];

In addition, you can create unmanaged arrays of CTS built-in value types by using the __nogc keyword. The syntax for declaring an unmanaged array of CTS built-in value types is type array_name __nogc[elements];. The syntax for declaring an unmanaged dynamic array of CTS built-in value types is type array_name __nogc = new type __nogc[elements];. For instance, the following statement declares an array of Int32 types on the unmanaged stack:

    Int32 arInvestments __nogc[10];

In contrast, the following statement shows how to declare a dynamic array of Int32 types on the unmanaged heap:

    Int32 arInvestments[] __nogc = new Int32 __nogc[10];

Managed Multidimensional Arrays

The syntax for creating managed multidimensional arrays differs somewhat from how you create traditional C++ multidimensional arrays. As with one-dimension arrays, when you create a multidimensional array using a CTS built-in value type, you must declare the array as dynamic. You must also use the syntax type array_name[ , , , ] = new type[dimension1, dimension2, ...];. You designate the number of dimensions in the array by the number of commas within the brackets following the array name. To be clear, you should place only commas within the brackets following the array name, nothing else. The number of dimensions is equal to the number of commas, plus one. Finally, you designate the number of elements for each dimension in the brackets attached to the data type at the end of the statement. For instance, the following statement declares a managed multidimensional array of the Double date type with two dimensions. The first dimension consists of six elements, and the second dimension consists of four elements:

    Double arTaxTable[,] = new Double[3,2];

You assign values to the elements of a managed multidimensional array in the same manner that you assign values to the elements of an unmanaged array. The following statement shows how to assign a value 63550 to the second column of the third row in the arTaxTable[] array:

    arTaxTable[2,1] = 63550;

Each of the garments that you added to the arGarments[] array needs to have an associated cleaning cost. For the DryCleaning class, you will create another one-dimension array named arDryCleaningCosts[], which stores the cost of cleaning each garment in the corresponding element in the arGarments[] array.
To add to the DryCleaning class a one-dimension array named arDryCleaningCosts[]:

1. Open the **DryCleaning.h** file in the Code Editor window.
2. In a new private section, declare an arDryCleaningCosts[] array of the Double data type, using the statement `Double arDryCleaningCosts[];`.
3. Next, open the **DryCleaning.cpp** file in the Code Editor window.
4. Add to the constructor function the statements shown in Figure 14-28, which initialize the arDryCleaningCosts[] array to ten elements and assign values to each element.

```
    arDryCleaningCosts[0] = 4.74;
    arDryCleaningCosts[1] = 5.46;
    arDryCleaningCosts[2] = 7.37;
    arDryCleaningCosts[3] = 5.05;
    arDryCleaningCosts[4] = 4.74;
    arDryCleaningCosts[5] = 4.12;
    arDryCleaningCosts[6] = 5.05;
    arDryCleaningCosts[8] = 4.74;
    arDryCleaningCosts[9] = 4.50;
```

**Figure 14-28**  arDryCleaningCosts[] array initialized in the DryCleaning class constructor function

For the Laundry class, you will create a two-dimensional array named arLaundryCosts[] that also stores the cost of cleaning each garment in the corresponding element in the arGarments[] array. However, the first column in the arLaundryCosts[] array will store the cost of cleaning and hanging each garment, and the second column will store the cost of cleaning and folding each garment.

To add to the Laundry class a two-dimensional array named arLaundryCosts[]:

1. Open the **Laundry.h** file in the Code Editor window.
2. Declare in a new private section an arLaundryCosts[] array of the Double data type with two dimensions, using the statement `Double arLaundryCosts[,] ;`.
3. Next, open the **Laundry.cpp** file in the Code Editor window.
4. Add to the constructor function the statements shown in Figure 14-29, which initialize the arLaundryCosts[] array to four elements in each dimension and assign values to the elements in each dimension.
Next, you will add member functions to the DryCleaning and Laundry classes that return values from individual array elements.

To add member functions to the DryCleaning and Laundry classes that return values from individual array elements:

1. Open the `DryCleaning.h` file in the Code Editor window and add to the public section the following declaration for the `getDryCleaningCost()` function:

   ```cpp
   Double getDryCleaningCost(Int32);
   ```

2. Now, open the `DryCleaning.cpp` file in the Code Editor window and add to the end of the file the following definition for the `getDryCleaningCost()` member function. The `getDryCleaningCost()` function returns the value in an element of the `arDryCleaningCosts[]` array using an `Int32` parameter named `iElement`:

   ```cpp
   Double DryCleaning::getDryCleaningCost(Int32 iElement) {
     return arDryCleaningCosts[iElement];
   }
   ```

3. Next, open the `Laundry.h` file in the Code Editor window and add to the public section the following declaration for the `getLaundryCost()` function:

   ```cpp
   Double getLaundryCost(Int32, Int32);
   ```

4. Finally, open the `Laundry.cpp` file in the Code Editor window and add to the end of the file the following definition for the `getLaundryCost()` member function. The `getLaundryCost()` member function returns the value in an element of the two-dimensional `arLaundryCosts[]` array, using two `Int32` parameters named `iElement0` and `iElement1`:

   ```cpp
   Double Laundry::getLaundryCost(Int32 iElement0, Int32 iElement1) {
     return arLaundryCosts[iElement0, iElement1];
   }
   ```

**Figure 14-29** `arLaundryCosts[]` array initialized in the Laundry class constructor function

Add these statements
The System::Array Class

So far, you have only manually written code to perform operations on unmanaged arrays. For instance, to find a specific value in an array, you need to write a looping statement that iterates through the array elements until it finds the correct value. Managed arrays, however, derive from the System::Array class, which contains a number of useful methods and properties for working with arrays. Figure 14-30 lists common methods of the System::Array class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear()</td>
<td>Sets the element values in all or part of an array to zero, false, or null, depending on the element type</td>
</tr>
<tr>
<td>Clone()</td>
<td>Creates a reference to an existing array</td>
</tr>
<tr>
<td>Copy()</td>
<td>Copies all or part of one array to another array, performing any necessary boxing or type casting</td>
</tr>
<tr>
<td>CopyTo()</td>
<td>Copies all or part of a one-dimensional array to another array</td>
</tr>
<tr>
<td>GetLength()</td>
<td>Returns the number of elements in a specified array dimension</td>
</tr>
<tr>
<td>GetValue()</td>
<td>Returns the value on a specified array element</td>
</tr>
<tr>
<td>IndexOf()</td>
<td>Returns the index number of the first occurrence of a specified value in an array or designated part of an array</td>
</tr>
<tr>
<td>LastIndexOf()</td>
<td>Returns the index number of the last occurrence of a specified value in an array or designated part of an array</td>
</tr>
<tr>
<td>Reverse()</td>
<td>Reverses the order of the elements in all or part of a one-dimensional array</td>
</tr>
<tr>
<td>SetValue()</td>
<td>Sets the value of a specified array element</td>
</tr>
<tr>
<td>Sort()</td>
<td>Sorts the elements in all or part of a one-dimensional array</td>
</tr>
</tbody>
</table>

**Figure 14-30** Common methods of the System::Array class

As an example of how to use the methods of the System::Array class, let us examine the Sort() and Reverse() methods. The Sort() method performs an alphanumerical sort of the elements in all or part of a one-dimensional array, using the syntax `Array::Sort(array_name);`. The following statements declare and sort a one-dimensional array of String* objects containing the names of departments in a hospital:

```c++
String* arDepts[] = new String*[8];
arDepts[0] = "Psychiatry";
arDepts[1] = "Ophthalmology";
arDepts[2] = "Biology";
arDepts[3] = "Neurology";
arDepts[4] = "Radiology";
arDepts[5] = "Otolaryngology";
arDepts[7] = "Pulmonary";
Array::Sort(arDepts);
```
for(Int32 i=0; i < 8; ++i) {
    Console::WriteLine(arDepts[i]);
}

If you were to execute the preceding code, the hospital department names would be listed in alphabetical order, as follows:

Biology
Neurology
Ophthalmology
Otolaryngology
Pediatrics
Psychiatry
Pediatrics
Radiology

The Reverse() method does not perform a reverse alphanumeric sort. Rather, it simply reverses the current order of the elements in an array. For instance, the following statements declare and reverse the order of elements in the arDepts[] array:

String* arDepts[] = new String*[8];
arDepts[0] = "Psychiatry";
arDepts[1] = "Ophthalmology";
arDepts[2] = "Biology";
arDepts[3] = "Neurology";
arDepts[4] = "Radiology";
arDepts[5] = "Otolaryngology";
arDepts[7] = "Pulmonary";
Array::Reverse(arDepts);
for(Int32 i=0; i < 8; ++i) {
    Console::WriteLine(arDepts[i]);
}

If you were to execute the preceding code, the hospital department names would be listed as follows. Notice that the order of hospital departments is simply the reverse of the order in which they were assigned to the elements of the arDepts[] array.

Pulmonary
Pediatrics
Otolaryngology
Radiology
Neurology
Biology
Ophthalmology
Psychiatry

Both the Sort() and Reverse() methods have overloaded forms that allow you to sort or reverse a specified range of elements within an array. See the entry for each method in the MSDN Library for more information.
One particularly useful property of the System::Array class is the **length property**, which returns the total number of elements in all of the dimensions of an array. You can append the length property to an array name using the syntax `array_name->length`. The following code prints the value 8 by using the length property to return the number of elements in the `arDepts[]` array:

```
String* arDepts[] = new String*[8];
arDepts[0] = "Psychiatry";
arDepts[1] = "Ophthalmology";
arDepts[2] = "Biology";
arDepts[3] = "Neurology";
arDepts[4] = "Radiology";
arDepts[5] = "Otolaryngology";
arDepts[7] = "Pulmonary";
Console::WriteLine(arDepts->Length);
```

You can use the GetLength() method of the System::Array class to return just the number of elements in a single dimension of a multidimensional array.

Next, you will add a member function named `garmentListMenu()` to the GarmentCleaning base class. The `garmentListMenu()` function prints to the console window a menu of the values assigned to the `arGarments[]` function. Because the GarmentCleaning base class does not know at run time how many elements are available in the `arGarments[]` function (because it is dynamically initialized in the derived Laundry and DryCleaning classes), you will use the length property of the System::Array class to find the number of elements in the array.

To add a member function to the GarmentCleaning base class that prints to the console window the values assigned to the `arGarments[]` function:

1. Open the `GarmentCleaning.h` file in the Code Editor window and add to the public section the following declaration for the `garmentListMenu()` function:

   ```cpp
   void garmentListMenu();
   ```

2. Now, open the `GarmentCleaning.cpp` file in the Code Editor window and add to the end of the file the following definition for the `garmentListMenu()` member function. The `garmentListMenu()` function uses the length property of the System::Array class to determine how many elements are stored in the dynamic array.

   ```cpp
   void GarmentCleaning::garmentListMenu() {
     for(Int32 i=0; i < arGarments->Length; ++i) {
       Console::Write(i + 1);
       Console::Write(".");
       Console::WriteLine(arGarments[i]);
     }
   }
   ```
Now you will add the rest of the functionality to the LaundryOrder() and DryCleaningOrder() functions in the Cleaners program’s main.cpp file.

To add the rest of the functionality to the LaundryOrder() and DryCleaningOrder() functions in the Cleaners program’s main.cpp file:

1. Open the main.cpp file in the Code Editor window.

2. Add the following code to the end of the LaundryOrder() function, just above its closing brace. The new code prompts the user to select the type of garment and whether it should be hung or folded. The for loop continues looping for however many items the user entered when he or she was prompted in the main() function. The for loop displays the garment list, using the garmentListMenu() function from the GarmentCleaning base class. After determining the type of garment and whether it should be hung or folded, the for loop retrieves the cost of cleaning the garment by calling the getLaundryCost() member function.

```cpp
String* pResponse;
Int32 iItem;
Int32 iHangerOrFolded;
Double dCost;
for(Int32 i=1; i<=iCount; ++i) {
    Console::WriteLine();
    pLaundry->garmentListMenu();
    Console::WriteLine();
    Console::Write("Select the garment type for item ");
    Console::Write(i);
    Console::Write(" ");
    pResponse = Console::ReadLine();
    iItem = Convert::ToInt32(pResponse);
    Console::WriteLine();
    Console::WriteLine("1. Hanger");
    Console::WriteLine("2. Folded");
    Console::WriteLine();
    Console::Write("Hanger or folded: ");
    pResponse = Console::ReadLine();
    iHangerOrFolded = Convert::ToInt32(pResponse);
    Console::WriteLine("Laundering cost: ");
    if (iHangerOrFolded == 1)
        dCost = pLaundry->getLaundryCost(0, iItem-1);
    else if (iHangerOrFolded == 2)
        dCost = pLaundry->getLaundryCost(1, iItem-1);
    Console::WriteLine(dCost);
}
```

3. Next, add the following code to the end of the DryCleaningOrder() function, just above its closing brace. The code is similar to the code you added in
Step 2, except that it is simpler because the DryCleaning class does not use a multidimensional array.

```cpp
String* pResponse;
Int32 iItem;
Double dCost;
for(Int32 i=1; i<=iCount; ++i) {
    Console::WriteLine();
    pDryCleaning->garmentListMenu();
    Console::WriteLine();
    Console::Write("Select the garment type for item ");
    Console::Write(i);
    Console::Write(" ");
    pResponse = Console::ReadLine();
    iItem = Convert::ToInt32(pResponse);
    Console::WriteLine();
    Console::Write("Laundering cost: ");
    dCost = pDryCleaning->getDryCleaningCost(iItem-1);
    Console::WriteLine(dCost);
}
```

4. Build and execute the program. Figure 14-31 shows the output after selecting the Laundry option.

![Output of Cleaners program after adding code to the LaundryOrder() and DryCleaningOrder() functions](image)

Figure 14-31 Output of Cleaners program after adding code to the LaundryOrder() and DryCleaningOrder() functions

5. Press any key to close the console window.

The .NET Framework class library introduces another very useful array class named ArrayList, which is defined in the System::Collections namespace. Arrays created from ArrayList have two advantages over arrays created with the Array class. First, they are capable of growing and shrinking their number of elements dynamically after the array is created. Second, you can store multiple data types within the elements of an ArrayList array.
MANAGED EXTENSIONS AND INHERITANCE

You must follow several rules when working with managed types and inheritance. One of the most important rules is that a managed class (both gc and value types) cannot inherit from an unmanaged class. Additionally, a managed class cannot inherit from more than one managed class. You also cannot derive an unmanaged class from a managed class. There are additional rules you need to follow when creating managed classes, but these are the major rules that apply to inheritance.

The __gc Classes and Value Classes topics in the MSDN Library contain complete lists of rules that apply to both types of managed classes.

Managed Extensions for C++ also include two additional keywords that apply to inheritance: the __abstract and __sealed keywords.

Abstract Classes

Recall that in unmanaged C++, you create an abstract class by including one or more pure virtual functions in the class definition. In managed C++, you create an abstract class by adding the __abstract keyword to a class declaration. Recall from Chapter 7 that you created an abstract Cultural base class. To declare the Cultural base class as a managed abstract class, you add the __abstract keyword to the class declaration, as follows:

```cpp
__abstract class Cultural : public Anthropology {
    public:
        Cultural(void);
```

Keep in mind that declaring a class with __abstract does not make it a managed class. One requirement for both gc types and value types is that they cannot inherit from unmanaged classes. Because the only thing you can do with an abstract class is use it as a base class, you must also use the __gc keyword in the declaration for a managed abstract class. In order to make the Cultural class both abstract and a gc type, you must use both the __abstract and __gc keywords in the class declaration, as follows:

```cpp
__abstract __gc class Cultural : public Anthropology {
    public:
        Cultural(void);
```

The __abstract keyword cannot be applied with the __value or __sealed keywords.

Next, you will use the __abstract keyword to turn the GarmentCleaning class into an abstract class.
To use the `__abstract` keyword to turn the GarmentCleaning class into an abstract class:

1. Open the `GarmentCleaning.h` file in the Code Editor window.
2. Add the `__abstract` keyword to the class header declaration so it reads as follows:
   ```cpp
   __abstract __gc class GarmentCleaning
   ```
3. Build and execute the program. It should function the same as it did before you created the abstract class.
4. Press any key to close the console window.

Sealed Classes

A **sealed class** is a class that cannot be used as a base class. You seal a class when it contains proprietary code or information that should not be modified in any derived classes. You create a sealed class by adding the `__sealed` keyword to a class declaration. The `__sealed` keyword can only be applied to a gc type, so you must use both `__sealed` and `__gc` in a class declaration, as follows:

```cpp
__sealed __gc class Savings {
    public:
        Savings(void);
}
```

The `__sealed` keyword can also be applied to virtual functions to prevent other programmers from overriding them.

Next, you will use the `__sealed` keyword to turn the Laundry and DryCleaning classes into sealed classes.

To use the `__sealed` keyword to turn the Laundry and DryCleaning classes into sealed classes:

1. Open the `Laundry.h` file in the Code Editor window, and add the `__sealed` keyword to the class header declaration so it reads as follows:
   ```cpp
   __sealed __gc class Laundry :
       public GarmentCleaning
   ```
2. Open the `DryCleaning.h` file in the Code Editor window, and add the `__sealed` keyword to the class header declaration so it reads as follows:
   ```cpp
   __sealed __gc class DryCleaning :
       public GarmentCleaning
   ```
3. Build and execute the program. It should function that same as it did before you created the sealed classes.
4. Press any key to close the console window.
Chapter 5 covered the topic of information hiding in great detail. As you learned, accessor functions are public member functions that a client can call to retrieve or modify the value of a data member. Because accessor functions often begin with the words get or set, they are also referred to as get or set functions.

The .NET Framework takes the concept of information hiding one step further through the use of properties. Early on in this book, you learned that properties are variables associated with an instantiated object. Examples of the properties of an object instantiated from a class named Payroll may include an employee’s number of tax withholding allowances, federal and state tax percentages, and the cost of insurance premiums. If the Payroll class includes a withholding allowances property, you could modify the property using the following statements:

```csharp
Payroll* januaryPayroll = new Payroll;
januaryPayroll->allowances = 5;
```

Properties, however, seem to go against the concept of information hiding because, technically, you should be able to access them only through accessor functions. But as you can see in the preceding example, the allowances property is being accessed directly, without an accessor function. The .NET Framework allows you to create properties that clients of your code can access as though they were data members. However, the values you assign to properties are really written to private data members through get and set functions, which is consistent with the concept of information hiding.

There are two types of properties: scalar and indexed. Scalar properties use typical get and set functions to retrieve and store values in an associated private data member. Indexed properties also use typical get and set functions, but allow clients to use array brackets to retrieve and access property values. For instance, you may have a class named Students that contains an Int32 array named arStudentID, which stores student IDs. Using an indexed property, you could write a property named StudentName that retrieves and stores the name of a student associated with a particular ID, using a statement such as `curStudent->StudentName[246];` The use of indexed properties is too advanced for the purposes of this chapter, so you will focus instead on scalar properties.

You create a scalar property by writing get and set functions in a class, the same type of get and set functions you wrote in earlier chapters. However, you also need to follow these rules:

- Both function declarations must begin with the `__property` keyword.
- The name of the get() function must begin with `get_` and the name of the set() function must begin with `set_`.
The get function must have no parameters and must return a value of the same data type as the associated private data member.

The set function must accept a single parameter of the same data type as the associated private data member.

Within the get function definition, you return the value of the private data member. Within the set function definition, you set the value of the private data member using the single parameter accepted by the function. Figure 14-32 shows a managed Stocks class that declares get and set functions using the __property keyword that retrieve and set the value of a private pStockName data member.

The text in the accessor function name that follows the words get_ and set_ becomes the name of the property. For instance, in the Stocks class, the property name is StockName. Clients of the Stocks class can now read and write the value of the StockName property directly; the Common Library Runtime will automatically call the accessor functions, which handle the actual writing and reading of data to and from the private pStockName data member. For instance, the following code assigns the value to and retrieves the value from the StockName property:

```c++
Stocks* curStock = new Stocks;
curStock->StockName = S"Cisco";
Console::WriteLine(curStock->StockName);
```
You cannot use the name of an existing data member of the class as the text that follows the `get_` and `set_` portions of the accessor function names. Doing so will result in a redefinition error because the compiler uses the property names in the accessor functions as if they were real data members. Common practice, however, is to use either a different letter case from the name of the actual data member or to use Hungarian notation only in the data member declaration (as was done in Figure 14-32).

Excluding the set function makes the property read-only, whereas excluding the get function makes the property write-only.

Next, you will add an OrderTotal property to the Cleaners program that stores a running total of the cost of each cleaning order. The OrderTotal property will read from and write to a double data member named dTotal.

To add an OrderTotal property to the Cleaners program:

1. Open `GarmentCleaning.h` in the Code Editor window.

2. As shown in Figure 14-33, add the declarations for the property’s accessor functions to the end of the public section. Also, add a declaration to the protected section for the double data member named dTotal.

3. Open `GarmentCleaning.cpp` in the Code Editor window and add to the end of the file the following definitions for the OrderTotal property’s accessor functions:

```cpp
void GarmentCleaning::set_OrderTotal(Double dOrderTotal) {
    dTotal = dOrderTotal;
}
```
To complete the Cleaners program, you will now add code to the LaundryOrder() and DryCleaningOrder() functions that writes to and reads from the OrderTotal property.

To add code to the LaundryOrder() and DryCleaningOrder() functions that writes to and reads from the OrderTotal property:

1. Open the main.cpp file in the Code Editor window.
2. As shown in Figure 14-34, add a statement to the end of the for loop in the LaundryOrder() function that adds the value of the dCost variable (which contains the cost of the current item to be cleaned) to the OrderTotal property.
3. Also as shown in Figure 14-34, add statements after the for loop that use the OrderTotal property to retrieve the total cost of the current order.
4. As shown in Figure 14-35, add a statement to the end of the for loop in the DryCleaningOrder() function that adds the value of the dCost variable (which contains the cost of the current item to be cleaned) to the OrderTotal property.
5. Also as shown in Figure 14-35, add statements after the for loop that use the OrderTotal property to retrieve the total cost of the current order.
6. Build and execute the program. Figure 14-36 shows the output after selecting all of the garment types for a dry-cleaning order.
CREATING WEB SERVICES

You created the Cleaners program as a console application in order to learn the techniques for writing managed code for the .NET Framework in an environment with which you are familiar. However, as mentioned earlier, managed C++ code is usually used for writing the server portion of a Web application or for writing Web services. Although a managed console application executes on the Common Language Runtime, it does not demonstrate how to access managed C++ code from the Web. The challenge you face is that accessing from the Web managed C++ code that exists on a server requires an understanding of Internet protocols such as HTTP, XML, and SOAP, along
with a strong knowledge of ASP .NET or another language that is capable of calling and executing the managed C++ server code and displaying any results in an HTML document loaded in a Web browser. Providing a detailed discussion of each of these technologies is beyond the scope of this chapter. Instead, this section will simply demonstrate how to quickly create a C++ Web service using the Managed C++ Web Service Wizard. Once you create the Web service, you will use the service help page to access the Web service in Internet Explorer. The **service help page** is a Web page that provides information about the functionality of a Web service and is useful if you want to quickly test a Web service.

In order to create Web applications and services, you must have Internet Information Server and FrontPage Server Extensions installed on your computer. By default, both of these items are selected when you first install Visual Studio .NET. However, if you chose not to install these items when you first installed Visual Studio .NET, then you will not be able to complete the exercises in this section. If you did not install Internet Information Server and FrontPage Server Extensions when you first installed Visual Studio .NET, see the article “Visual Studio .NET Software Requirements” in the MSDN Library for important configuration steps you must follow in order to add Internet Information Server and FrontPage Server Extensions to your existing Visual Studio .NET installation.

The main classes for creating a Web service are located in the System::Web::Services namespace in the .NET Framework class library. To give a Web service access to the System::Web::Services namespace, you must use the following #using statement, which imports the System.Web.Services.dll file:

```csharp
#using <System.Web.Services.dll>
```

One class of the System::Web::Services namespace that you will use in this section is the WebMethodAttribute class. The WebMethodAttribute class creates an attribute that marks a function as being callable by clients of the Web service. You mark a function with an attribute such as the WebMethodAttribute class simply by placing the attribute statement immediately above the function declaration statement.

When you use the Managed C++ Web Service Wizard to create a Web service, it creates a header file with an extension of .h and an implementation file with an extension of .cpp. However, the header and implementation files will reflect the name of the Web service, not the name of a class. Within the Web service’s header and implementation files you will see the declaration and definition for a class named Class1. Generally, you should change the name of the Class1 class to something more descriptive of the Web service. The Class1 class includes a single function named HelloWorld() that returns the string “Hello World.”
The HelloWorld() function definition is marked with the WebMethodAttribute class so that it is callable as a method by clients of the Web service, as follows:

```
[System::Web::Services::WebMethod]
String __gc* HelloWorld();
```

The preceding code uses the name WebMethod, which is an alias for the WebMethodAttribute class. By default, the Managed C++ Web Service Wizard uses the name WebMethod instead of WebMethodAttribute, although you are free to change it if you like.

You add a description to a Web service method by adding a set of parentheses containing Description=string following the WebMethod class name. For instance, to add a description to the HelloWorld() function that reads Returns the string ‘Hello, World!’, you modify the WebMethod attribute as follows:

```
[System::Web::Services::WebMethod(Description="Returns the string 'Hello, World!'")] String __gc* HelloWorld();
```

After you create the Web service, you will notice several new files in the Solution Explorer window. One file you need to be aware of is a file with an extension of .asmx. The .asmx file tells the server which class contains the implementation of the Web service. For example, if you open the .asmx file for a Web service named Currency, you will see the following statement, which specifies to the server that the Class1 class contains the implementation of the Currency Web service:

```
<%@ WebService Class=Currency.Class1 %>
```

If you change the name of the Class1 class to something more descriptive, then you must also change the class name referenced in the .asmx file. For instance, a more descriptive name for the Currency Web service’s class may be Exchange. Therefore, you should change the .asmx file to read as follows:

```
<%@ WebService Class=Currency.Exchange %>
```

Next, you will create a simple Web service that converts miles to kilometers and kilometers to miles.

To create a simple Web service that converts miles to kilometers and kilometers to miles:

1. Create a new project named Distance, using the Managed C++ Web Service Wizard. Save the project in the Chapter.14 folder in your Visual C++ Projects folder. After you click the OK button, the wizard immediately opens the new project in the IDE.

2. Open the Distance.h file in the Code Editor window. You will see the #using statement that gives the Web service access to the System.Web.Services.dll file. You will also see several using statements that give the Web service access to
several namespaces in the .NET Framework class library. If you scroll down in the file, you will see that the wizard created a Distance namespace to manage the Web service.

3. Locate the header declaration for the Class1 class. Notice that it is declared as a gc type using the __gc keyword. Also notice that the class extends the WebService base class. Change the Class1 class name to Conversion so the class header declaration reads as follows:

```c++
public __gc

class Conversion : public WebService
```

4. Next, replace all of the comments and statements in the public section with the two function declarations shown in Figure 14-37. The milesToKilometers() function converts miles to kilometers, and the kilometersToMiles() function converts kilometers to miles. Notice that each function is marked with the WebMethod attribute and includes a description.

5. Open the Distance.asmx file in the Code Editor window and modify the statement so it references the Conversion class instead of the Class1 class, as shown below:

```c_xml
<%@ WebService Class=Distance.Conversion %>
```

6. Now open the Distance.cpp file in the Code Editor window and replace all of the statements in the Distance namespace with the definitions shown in Figure 14-38 for the milesToKilometers() and kilometersToMiles() functions.

7. Build the project, but do not execute it.
When you build a Web service, Visual C++ not only compiles and links the various files that make up the service, it also deploys the files to your default home directory. Your home directory is the central location for your Web pages. The default home directory for Internet Information Server is C:\Inetpub\wwwroot. From a Web browser, you access the Web pages in your home directory using the URL http://localhost/, followed by a directory name and Web page you want to open. If you successfully built the project in the preceding exercise, then you should see a new Distance folder in your C:\Inetpub\wwwroot folder. One of the files in the Distance folder is named Distance.asmx. Opening a file with an extension of .asmx launches a service help page that provides information about the associated Web service and allows you to test its methods. Next, you will open the Distance.asmx file in Internet Explorer in order to test the Distance Web service.

To open the Distance.asmx file in Internet Explorer and test the Distance Web service:

1. Open Internet Explorer, type http://localhost/Distance/Distance.asmx into the Address box, and then press Enter or click the Go button. As shown in Figure 14-39, the service help page for the Distance Web service opens in the browser window. The two bullets at the top of the page are links that allow you to test each of the Web service’s methods. Notice that the descriptions you added to the WebMethod attributes appear beneath the name of each method. The rest of the page contains information on working with namespaces in Web services.

2. Click the kilometersToMiles link. A new page, shown in Figure 14-40, opens that allows you to test the kilometersToMiles() method. You use the Value text box to pass an appropriate argument to the function, which you can then test by clicking the Invoke button. The rest of the page contains information on working with the various Internet protocols that are used by Web services.
3. Enter a number in the **Value** text box, and then click the **Invoke** button to test the `kilometersToMiles()` method. Clicking the Invoke button opens a new Web browser window containing the results returned from the function. The function does not simply return a number that is the result of the
calculation. Instead, it returns the result within an XML document. Figure 14-41 shows the XML document returned from the kilometersToMiles() method after passing to the method a value of 10. The converted value is 6, which appears in bold in the figure.

![XML document](attachment:image.jpg)

**Figure 14-41** Results returned from the kilometersToMiles() method

XML, or Extensible Markup Language, is a standard method for exchanging data across the Internet.

4. Close both Web browser windows.

This chapter presents only the tip of the iceberg when it comes to writing managed C++ code for the .NET Framework. This section in particular showed only the most basic method of creating a Web application. However, the goal of this chapter was simply to get you started with .NET Framework programming. Now it’s up to you to build on what you have learned.

Your goal in the study of programming, or of any technology subject, for that matter, should not be memorizing facts and syntax. Your goal should be comprehending and understanding how things work. If you forget everything else you learned in this text, remember this: The best programmers in the world do not necessarily know all the answers. Rather, they know where to find all the answers. Use the MSDN Library to further your understanding of C++, MFCs, and the .NET Framework, and build yourself a library of reference books that you can use to find the answers you need.
CHAPTER SUMMARY

- The .NET Framework is a platform designed for creating, deploying, and running Web applications and services.
- A Web application is a program that executes on a server, but that clients access through an HTML document loaded in a Web browser.
- A Web service, or XML Web service, is a software component that resides on a Web server.
- A run-time host is a software component capable of loading and executing a specific type of .NET Framework application.
- Code that you write for the .NET Framework, regardless of the programming language, is called managed code. Code that is not written for the .NET Framework is called unmanaged code.
- Applications written for the .NET Framework are called managed applications, whereas applications that are not written for the .NET Framework are called unmanaged applications.
- Managed code executes in the Common Language Runtime, or CLR, which is the run-time platform for the .NET Framework.
- When you compile managed code for the Common Language Runtime, the code is first compiled into a special assembly language called Microsoft Intermediate Language, or MSIL.
- Portable executable (PE) file is the file format for executable programs or for files that will be assembled into executable programs.
- The .NET Framework uses assemblies as the basic unit of construction, deployment, versioning, and security.
- Metadata is information that describes individual elements managed by the Common Language Runtime.
- In order to provide cross-language compatibility, the Common Language Runtime uses the Common Type System (CTS) to define how types are declared, used, and managed.
- The Common Language Specification (CLS) defines a set of basic language rules to which all programming languages that target the Common Language Runtime must adhere.
- The .NET Framework class library is a large collection of classes used for writing programs that conform to the .NET Framework.
- Managed extensions are new keywords that have been added to C++ that allow you to write managed applications.
The Common Language Runtime garbage collector allocates and deallocates memory on a managed heap for .NET Framework applications.

Value types are stored on the managed stack and are used for small classes with short lifetimes that do not need to be allocated dynamically.

A gc (for “garbage collected”) type, or reference type, is stored on the managed heap and is managed by the garbage collector.

A pointer that points to a memory address on the managed heap is called a managed pointer or gc pointer.

You can use the __pin keyword to prevent the object of a pointer from being moved by the Common Language Runtime during garbage collection.

A managed array is stored on the managed heap and is controlled by the garbage collector.

In managed C++, you create an abstract class by adding the __abstract keyword to a class declaration.

A sealed class is a class that cannot be used as a base class. You create a sealed class by adding the __sealed keyword to a class declaration.

Scalar properties use typical get and set functions to retrieve and store values in an associated private data member.

Indexed properties use get and set functions that allow clients to use array brackets to retrieve and access property values.

**Review Questions**

1. A(n) _____________ is a software component capable of loading and executing a specific type of .NET Framework application.
   a. virtual machine
   b. run-time host
   c. execution environment
   d. fundamental platform

2. How can you add Windows Form controls to a Visual C++ application?
   a. using the Window Forms Editor
   b. using the Dialog Editor
   c. using the Win32 API Editor
   d. using the Code Editor
3. Code that you write for the .NET Framework, regardless of the programming language, is referred to as __________ code.
   a. framework
   b. Common Language Runtime
   c. managed
   d. unmanaged

4. The run-time platform for the .NET Framework is called the __________.
   a. framework run-time environment
   b. Common Language Runtime
   c. framework application target
   d. virtual run-time platform

5. When you compile managed code for the Common Language Runtime, the code is first compiled into a special assembly language called __________.
   a. machine language
   b. bytecode
   c. scripting code
   d. Microsoft Intermediate Language

6. Which of the following is not a correct extension for a portable executable file?
   a. .dll
   b. .pe
   c. .obj
   d. .lib

7. Where is assembly information stored when you compile a .NET application?
   a. in a separate file with an extension of .lib
   b. in a separate file with an extension of .asi
   c. in the PE file
   d. Assembly information is not stored when you compile a .NET Framework application.

8. __________ is information that describes individual elements in a .NET Framework application.
   a. Metadata
   b. CLS
   c. MSIL
   d. CRL
9. What is the correct syntax for a predefined attribute named `AssemblyTitleAttribute`?
   a. `AssemblyTitleAttribute("Hello World");`
   b. `assembly:AssemblyTitleAttribute("Hello World");`
   c. `[assembly:AssemblyTitleAttribute = "Hello World"];`
   d. `[assembly:AssemblyTitleAttribute("Hello World")];`

10. What is the CTS equivalent of the primitive C++ int data type?
    a. SByte
    b. Int16
    c. Int32
    d. Int64

11. What is the CTS equivalent of the primitive C++ float data type?
    a. Single
    b. Double
    c. Float
    d. Floating

12. In order to ensure that components written for the .NET Framework are compatible with one another, they must conform to the______________.
    a. Java VM
    b. .NET Framework class library
    c. Common Library Runtime
    d. Common Language Specification

13. What is the correct statement for importing into a C++ program the core classes in the .NET Framework class library?
    a. `#include <mscorlib.dll>
    b. `#include "mscorlib.dll"
    c. `#using <mscorlib.dll>
    d. `#using "mscorlib.dll"

14. Many of the basic classes in the .NET Framework class library are stored in the ______________ namespace.
    a. System
    b. std
    c. Framework
    d. Common Language Runtime
15. Which .NET Framework class contains methods and properties for accessing the standard input and output streams?
   a. Stream
   b. IOSStream
   c. Console
   d. STDIO

16. What is the correct syntax to declare a .NET Framework String class variable?
   a. string sCompanyName = "Microsoft";
   b. String sCompanyName = "Microsoft";
   c. string* pCompanyName = "Microsoft";
   d. String* pCompanyName = "Microsoft";

17. A Unicode character occupies ____________ bytes.
   a. 0
   b. 1
   c. 2
   d. 4

18. Which of the following prefixes results in a single instance of the String class being created, and converts the string literal to Unicode?
   a. L
   b. U
   c. S
   d. T

19. Which method of the Convert class converts a String class variable to a CTS Double data type?
   a. ConvertToDouble()
   b. ConvertDouble()
   c. ToDouble()
   d. Double()

   a. garbage collector
   b. trash collector
   c. trash compactor
   d. memory manager
21. What keyword do you use to explicitly declare an unmanaged type?
   a. __gc
   b. __nogc
   c. __value
   d. __unmanaged

22. __________ types are stored on the managed stack and are used for small classes with short lifetimes that do not need to be allocated dynamically.
   a. Garbage
   b. Unmanaged
   c. Managed
   d. Value

23. Which keyword do you use to explicitly declare a garbage-collected type?
   a. __gc
   b. __nogc
   c. __value
   d. __unmanaged

24. When are you required to use the delete keyword to delete an object that is stored on the managed heap?
   a. when you no longer need the type
   b. before the program ends
   c. whenever your program needs more memory
   d. You are not required to use the delete keyword with a managed object.

25. Which keyword do you use to prevent the object of a managed pointer from being moved around in memory?
   a. __static
   b. __box
   c. __mem
   d. __pin

26. When you create a managed array using a CTS built-in value type, you must declare it on the __________.
   a. unmanaged heap
   b. unmanaged stack
   c. managed heap
   d. managed stack
27. What is the correct syntax for declaring a managed multidimensional array?
   a. Int32 arSales[3,2];
   b. Int32 arSales = new Int32[3,2];
   c. Int32 arSales[] = new Int32[3,2];
   d. Int32 arSales[,] = new Int32[3,2];

28. Which property of the System::Array class returns the total number of elements in all of the dimensions of an array?
   a. elements
   b. dimensions
   c. size
   d. length

29. Which keyword do you use to mark a managed type as abstract?
   a. __abstract
   b. __final
   c. __pin
   d. __virtual

30. Which keyword do you use to prevent a managed type from being used as a base class?
   a. __nobase
   b. __last
   c. __nogc
   d. __sealed

31. What is the name of the associated property for a managed accessor function named set_Company_Name()?
   a. Company
   b. Name
   c. Company_Name
   d. CompanyName

**Programming Exercises**

1. Modify the following type declarations to the common type system built-in value type equivalents:
   - char cData;
   - int iPeriod;
• short siMonths;
• long liProjectedFigures;
• float fWeeklySalary;
• double dGrossRevenue;
• bool bHealthBenefits;
• int ctLoop;

2. Modify the following program so that it is a managed application. Be sure to change the data types and output statements.

```c++
#include <iostream>
using namespace std;

void main() {
    cout << "New York City"
         << " is the greatest city"
         << " in the world'" << endl;
    int iBoroughs = 5;
    int iPopulation = 7322564;
    cout << "It has " << iBoroughs
         << " boroughs, with a population of approximately "
         << iPopulation << ".";
}
```

3. Modify the following code so that it uses the .NET Framework String class along with Console class output statements. Be sure to modify the string literals so that only a single instance of the String class is created.

```c++
#include <iostream>
#include <string>
using namespace std;

void main() {
    string sFirstClass = "Chemistry";
    string sSecondClass = "Algebra";
    string sThirdClass = "Spanish";
    cout << "Your first class is "
         << sFirstClass << endl;
    cout << "Your second class is "
         << sSecondClass << endl;
    cout << "Your third class is "
         << sThirdClass << endl;
}
```

4. Modify the following program so that it is a managed application. Use the Equals() method of the String class to evaluate the variable containing the name of the sport.

```c++
#include <iostream>
#include <string>
```
using namespace std;
void main() {
    string sSport = S"baseball";
    if (sSport == "golf")
        cout << "Golf is played on a golf course."
        << endl;
    else if (sSport == "tennis")
        cout << "Tennis is played on a tennis court."
        << endl;
    else if (sSport == "baseball")
        cout << "Baseball is played on a baseball diamond."
        << endl;
    else if (sSport == "basketball")
        cout << "Basketball is played on a basketball court."
        << endl;
    else
        cout << "I don't recognize your sport."
        << endl;
}

5. Modify the following program so that it is a managed application. Be sure to change the data types and output statements.
#include <iostream>
#include <string>
using namespace std;
void main() {
    int iCurrentAge;
    int iRetirementAge = 65;
    cout << "Enter your current age: ";
    cin >> iCurrentAge;
    cout << "You will be eligible for full "
    << "social security retirement benefits in "
    << iRetirementAge - iCurrentAge
    << " years" << endl;
}

6. Modify the following class declaration so that it is declared on the managed heap instead of the managed stack:
__value class MutualFund {
public:
    MutualFund();
...

7. Modify the following code so that the object of the pCurTemp pointer is not moved by the Common Language Runtime during garbage collection. Assume that the Temp class is managed on the run-time heap.
...
#include "temp.h";
void convertToCelsius(Temp*);
void main() {
    Temp* pCurTemp = new Temp;
    pCurTemp->setTemperature(55);
    convertToCelsius(pCurTemp);
}
void convertToCelsius(Temp* pTemp) {
    Double dResult =
    (pTemp->getTemperature() - 32)
    * .55;
    Console::Write(
    "The temperature you entered as Fahrenheit is equal to ");
    Console::Write(dResult);
    Console::WriteLine(" in Celsius");
}

8. Fix the following statement so that it will compile in a managed C++ application:
    Char arGrades = new Char[10];

9. Create a managed multidimensional Double array named arBoxes containing
    three rows and three columns. The three rows will represent a small box, a
    medium box, and a large box. The three columns will represent the length, width,
    and depth of each box. After declaring and initializing the array, write a for loop
    that prints the volume (length * width * depth) of each box. Create the array
    using the dimensions shown in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Length</th>
<th>Width</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small box</td>
<td>12</td>
<td>10</td>
<td>2.5</td>
</tr>
<tr>
<td>Medium box</td>
<td>30</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>Large box</td>
<td>60</td>
<td>40</td>
<td>11.5</td>
</tr>
</tbody>
</table>

10. Modify the following code so that it is a managed abstract base class:
    #pragma once
    #using <mscorlib.dll>
    using namespace System;
    class Inventory
    {
    public:
        Inventory(void);
        ~Inventory(void);
    };

11. Modify the following class declaration so that it cannot be used as a base class:
    __gc class Transportation {
    public:
        Transportation();
        ...
12. Modify the accessor functions in the following code so that the private data members are accessed as properties. Also, write the appropriate implementation file for the class.

```c++
#pragma once
#using <mscorlib.dll>
using namespace System;
__gc class MutualFund {
public:
    MutualFund(void);
~MutualFund(void);
    void setNumberOfShares(Int32);
    void setAnnualYield(Double);
    Int32 getNumberOfShares();
    Double getAnnualYield();
private:
    Int32 iNumberOfShares;
    Double dAnnualYield;
};
```

**Programming Projects**

1. Create a managed console application named ExecutiveSalaries. Think of some fictitious names for the top five executive positions at a corporation: chairman, chief executive officer, chief operating officer, chief information officer, and chief financial officer. Within the main() function, declare three variables for each of the executives; one variable should contain the executive’s name, another variable should contain the executive’s title, and the last variable should contain the executive’s salary. Assign each executive’s name, title, and salary to his or her respective variables, and then print all of the variables. Each individual executive’s name, title, and salary should be printed on its own line.

2. Write a managed console application named Rainfall that allows a user to enter the number of inches it rained during each month of the year. Assign each value to the corresponding element in a twelve-element array. Calculate the average yearly rainfall using the array elements, and then print the rainfall for each individual month to the screen, followed by the total yearly rainfall and then the average monthly rainfall.

3. Create a managed console application that contains an Automobile class. Include private data members such as make, model, color, and engine, along with the appropriate property functions for setting and retrieving private data members. Use Console class methods to gather and display information.

4. A painting company estimates the cost of its jobs based on materials and labor costs. The cost of materials is 15 cents per square foot; the cost of labor is 25 cents per square foot. Write a managed console application with a Painting
class that determines the cost of painting a house, by allowing prospective customers to enter the estimated number of feet they need to have painted. Store the number of feet in a private data member, along with property functions for setting and retrieving the value of the private data member. Use separate member functions for determining the cost of materials and the cost of labor. Also, include a function that calls the member functions for the materials and labor costs in order to calculate the total cost of the job. Store the total estimate in a private data member and print the estimate to the screen.

5. Write a managed console application named Discount. Declare in the main() function a Double array named arRetail[] that contains 5 elements. Assign to the arRetail[] array the following values: 99.5, 78.65, 32.40, 59.95, and 12.75. Write a custom function named discount() that accepts a one-dimension array. The discount() function should not return a value. Within the body of the discount() function, reduce by 10% the values of the array that are passed to it. (You reduce the value of a number by a specific percentage by multiplying the number itself by 1 plus the percentage amount. For example, to reduce the number 150 by 30% you use the statement 150 * 1.3.) Call the discount() function from the main() function and pass the arRetail[] array to it. Print the values of the arRetail[] array before and after the call to the discount() function. Be sure to use the Length property of the Array class to determine the number of elements in the arRetail[] array.

6. Write a Managed C++ Console Application named EnvironmentInfo that uses the Environment class in the System namespace to print information about your computer and operating system.

7. Write a Web service named Temperature that contains one function for converting Celsius to Fahrenheit and another function for converting Fahrenheit to Celsius. Refer to the Conversion Center program you created in Chapter 8 for the appropriate calculations. Add descriptions to the WebService attributes and modify the class name to something meaningful. Be sure to test your Web service using the service help page.