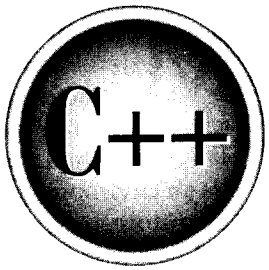
The Complete Reference



Appendix A

The .NET Managed Extensions to C++

Microsoft's .NET Framework defines an environment that supports the development and execution of highly-distributed, component-based applications. It enables differing computer languages to work together, and provides for security, program portability, and a common programming model for the Windows platform. Although the .NET Framework is a relatively recent addition to computing, it is an environment in which many C++ programmers will likely be working in the near future.

Microsoft's .NET Framework provides a managed environment that oversees program execution. A program targeted for the .NET Framework is not compiled into executable object code. Rather, it is compiled into Microsoft Intermediate Language (MSIL), which is then executed under the control of the Common Language Runtime (CLR). Managed execution is the mechanism that supports the key advantages offered by the .NET Framework.

To take advantage of .NET managed execution, it is necessary for a C++ program to use a set of nonstandard, extended keywords and preprocessor directives that have been defined by Microsoft. It is important to understand that these extensions are not defined by ANSI/ISO standard C++. Thus, code in which they are used is nonportable to other environments.

It is far beyond the scope of this book to describe the .NET Framework, or the C++ programming techniques necessary to utilize it. (A thorough explanation of the .NET Framework and how to create C++ code for it would easily fill a large book!) However, a brief synopsis of the .NET managed extensions to C++ is given here for the benefit of those programmers working in the .NET environment. A basic understanding of the .NET Framework is assumed.

The .NET Keyword Extensions

To support the .NET managed execution environment, Microsoft adds the following keywords to the C++ language:

__abstract	__box	__delegate
__event	__finally	__gc
__identifier	__interface	__nogc
__pin	__property	__sealed
__try_cast	__typeof	__value

Each of these is briefly described in the following sections.

__abstract

__abstract is used in conjunction with **__gc** to specify an abstract managed class. No object of an **__abstract** class can be created. A class specified as **__abstract** is not required to contain pure virtual functions.

__box

__box wraps a value within an object. Boxing enables a value type to be used by code that requires an object derived from **System::Object**, which is the base class of all .NET objects.

__delegate

__delegate specifies a delegate, which encapsulates a pointer to a function within a managed class (that is, a class modified by __gc).

__event

__event specifies a function that represents an event. Only the prototype for the function is specified.

__finally

__finally is an addition to the standard C++ exception handling mechanism. It is used to specify a block of code that will execute when a **try/catch** block is left. It does not matter what conditions cause the **try/catch** block to terminate. In all cases, the __finally block will be executed.

__gc

__gc specifies a managed class. Here, "gc" stands for "garbage collection" and indicates that objects of the class are automatically garbage collected when they are no longer needed. An object is no longer needed when no references to the object exist. Objects of a __gc class must be created using **new**. Arrays, pointers, and interfaces can also be specified as __gc.

__identifier

__identifier allows a C++ keyword to be used as an identifier. This is a special-purpose extension that will not be used by most programs.

__interface

__interface specifies a class that will act as an interface. In an interface, no function can include a body. All functions in an interface are implicitly pure virtual functions. Thus, an interface is essentially an abstract class in which no function has an implementation.

__nogc

__nogc specifies a nonmanaged class. Since this is the type of class created by default, the __nogc keyword is not usually used.

__pin

__pin is used to specify a pointer that fixes the location in memory of the object to which it points. Thus, an object that is "pinned" will not be moved in memory by the garbage collector. As a result, garbage collection does not invalidate a pointer modified by __pin.

__property

__property specifies a property, which is a member function that gets or sets the value of a member variable. Properties provide a convenient means to control access to private or protected data.

__sealed

__sealed prevents the class that it modifies from being inherited. It can also be used to specify that a virtual function cannot be overridden.

__try_cast

__try_cast attempts to cast one type of expression into another. If the cast fails, an exception of type **System::InvalidCastException** is thrown.

__typeof

__typeof obtains an object that encapsulates type information for a given type. This object is an instance of **System::Type**.

__value

__value specifies a class that is represented as a value type. A value type holds its own values. This differs from a __gc type, which must allocate storage through the use of **new**. Value types are not subject to garbage collection.

Preprocessor Extensions

To support .NET, Microsoft defines the **#using** preprocessor directive, which is used to import metadata into your program. Metadata contains type and member information in a form that is independent of a specific computer language. Thus, metadata helps support mixed-language programming. All managed C++ programs must import **<mscorlib.dll>**, which contains the metadata for the .NET Framework.

Microsoft defines two pragmas that relate to the .NET Framework. (Pragmas are used with the **#pragma** preprocessing directive.) The first is, **managed**, which specifies managed code. The second is **unmanaged**, which specifies unmanaged (that is, native) code. These pragmas can be used within a program to selectively create managed and unmanaged code.

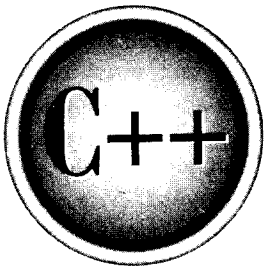
The attribute Attribute

Microsoft defines **attribute**, which is the attribute used to declare another attribute.

Compiling Managed C++

At the time of this writing, the only compiler commonly available that can target the .NET Framework is the one supplied by Microsoft's Visual Studio .NET. To compile a managed code program, you must use the **/clr** option, which targets code for the Common Language Runtime.

The Complete Reference



Appendix B

C++ and the Robotics Age

I have had a long term interest in robotics, especially robotic control languages. In fact, years ago I designed and implemented industrial robotic control languages for use on small educational robots. Although I no longer work professionally in the area of robotics, it remains an important and engaging special interest of mine. Over the years I have seen the capabilities of robots (and the code that controls them) make major leaps forward. We now stand at the beginning of the robotics age. There are already robots that can mow the lawn and vacuum the floor. They assemble our cars and work in environments dangerous to humans. Battlefield robots are now becoming a reality. Many more robotic applications are on the way. As robots become more commonplace, integrating themselves into the fabric of everyday life, increasing numbers of programmers will find themselves writing robotic control code. And, much of that code will be in C++.

C++ is a natural choice for robotic programming because robots require efficient, high-performance code. This is especially true for the low-level motor control routines, and for such things as vision processing, where speed is quite important. Although some parts of a robotic subsystem, such as a natural language processor, may be written in a language such as C#, the low-level code will almost certainly remain in C++. C++ and robotics go hand-in-hand.

If you are interested in robotics, especially if you are interested in creating your own robot for experimentation, then you might find the robot in Figure B-1 of interest. This is my current test robot. Several things make this robot interesting. First, it contains an

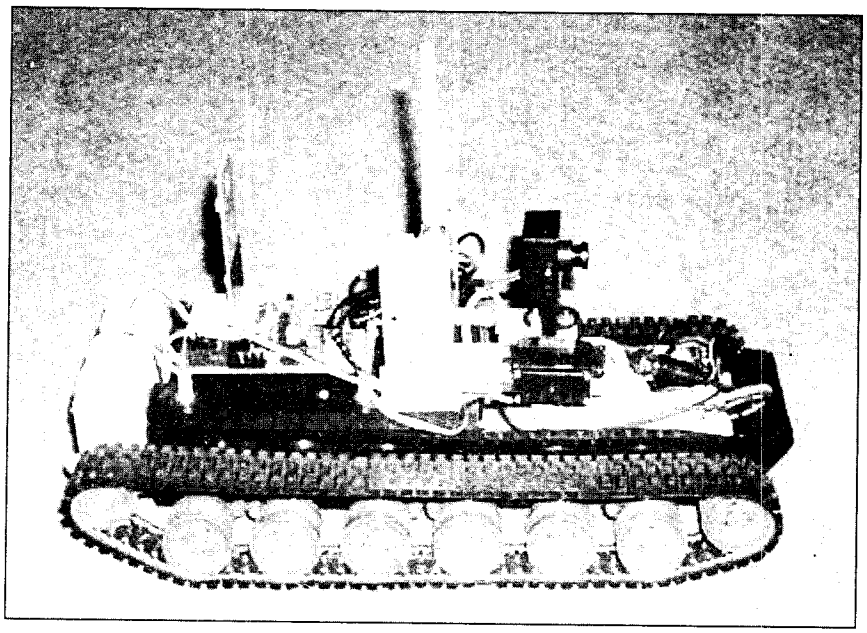


Figure B-1. *A simple, yet effective experimental robot (Photo by Ken Kaiser)*

on-board microprocessor that provides basic motor control and sensor feedback. Second, it contains an RS-232 transceiver that is used to receive instructions from the main computer and return results. This approach enables a remote computer to provide the intensive processing that is necessary in robotics without adding all that weight to the robot, itself. Third, it contains a video camera that is connected to a wireless video transmitter.

The robot is built on a Hobbico M1 Abrams R/C tank chassis. (I have found that the chassis of R/C model tanks and cars often work well as a robot base.) I removed most of the internals from the tank, including the receiver and speed controls, but I kept the motors. The Hobbico tank is well suited for a robotics platform because it is quite strong, the motors are good, it can carry a lot of weight, and its tank treads don't fall off. Also, by using tank treads, the robot has a zero turning radius and can run on uneven ground. The chassis is about 18 inches long and about 8 inches wide.

Once the chassis was empty, I added the following components. To provide on-board control, I used a BASIC Stamp 2, which is a simple, yet powerful microprocessor manufactured by Parallax. Inc. (www.parallaxinc.com). The RS-232 transceiver is also from Parallax, as is the video camera and transmitter. Both the wireless RS-232 transceiver and the video transmitter have a range of about 300 feet. I also added electronic speed controllers for the tank motors. They are of the type used by high-performance R/C cars. They are controlled by the BASIC Stamp microprocessor.

Here is the way the robot works. The remote computer runs the main robotic control program. This program handles all "heavy-duty" processing, such as vision, guidance, and spatial orientation. It can also learn a series of moves and then replay them. The remote computer transmits motion-control instructions (via the wireless RS-232 link) to the robot. The BASIC Stamp receives those instructions and puts them into action. For example, if a "move forward" command is received, the BASIC Stamp sends the proper signals to the electronic speed controllers connected to the motors. When the robot has completed a command, it returns an acknowledgement code. Thus, communication between the remote computer and the robot is bi-directional, and the successful completion of each command can be confirmed.

Because the main processing for the robot occurs on the remote computer, there are no severe limitations to the amount of processing that I can do. For example, at the time of this writing, the robot can follow an object by using its vision system. This capability requires a fair amount of processing that would be difficult to carry on board.

Recently, I have begun work on a robot arm that will be added to the robot. A prototype of the arm is shown in Figure B-2. Although there are several commercial robot arms available to the experimenter and hobbyist, I decided to create my own because I wanted an arm that would be stronger and able to lift heavier objects than was commonly available. The arm uses a stepper motor mounted at its base to turn a long, threaded screw which opens and closes the gripper. This approach allows precise movement along with considerable strength. The arm is controlled by its own Stamp. Thus, the main robot controller simply hands off arm commands to the second Stamp. This allows fully parallel operation of the robot and the arm, and prevents bogging down the main robot controller.

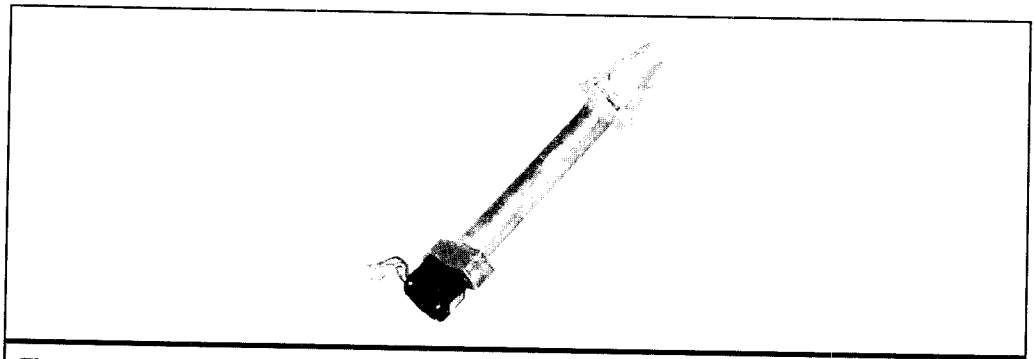


Figure B-2. *A prototype robot arm (Photo by Ken Kaiser)*

Although the main robotic control code will always remain in C++, I am experimenting with migrating a couple of subsystems, including the RS-232 communication routines, to C#. C# offers a convenient interface to IP data transfers and being able to control the robot from a remote location via the Internet is a tantalizing thought.