C# for C++ Developers

This appendix is intended for developers who are already familiar with C++ and want to see what the differences are between C++ and C#. It surveys the C# language, noting specifically those areas in which it is different from C++. Because the two languages do have a large amount of syntax and methodology in common, advanced C++ programmers may find they can use this appendix as a shortcut to learning C#.

It should be made clear that C# is a distinct language from C++. Whereas C++ was designed for general object-oriented programming in the days when the typical computer was a standalone machine running a command-line-based user interface, C# is designed specifically to work with .NET and is geared to the modern environment of Windows and mouse-controlled user interfaces, networks, and the Internet. There is a similarity between the two languages, particularly in syntax, and this is not surprising because C# was designed as an object-oriented language that took the good points of earlier object-oriented languages — of which C++ has been arguably the most successful example — but learned from the poorer design features of these languages

Because of the similarities between the two languages, developers who are fluent in C++ may find that the easiest way to learn C# is to treat it as C++ with a few differences and learn what those differences are. This appendix is designed to help you do that.

The appendix starts off with a broad overview, mentioning, in general terms, the main differences between the two languages, but also indicating the areas they have in common. This is followed by a comparison of what the standard Hello, World program looks like in each of the two languages. The bulk of this appendix is dedicated to a topic-by-topic analysis that looks at each of the main language areas and gives a detailed comparison between C# and C++; inevitably, an appendix of this size cannot be comprehensive, but it covers all the main differences between the languages that you will notice in the course of everyday programming. It is worth pointing out

that C# relies heavily on support from the .NET base class library in a large number of areas. This appendix is largely restricted to the C# language itself and does not extensively cover the base classes.

For the purposes of comparison, ANSI C++ is taken as a reference point. Microsoft has added numerous extensions to C++ (many new extensions with .NET 2.0), but these are not normally used in this appendix when comparing the two languages.

Conventions for This Appendix

Note that this appendix adopts an additional convention when displaying code; C# code is always displayed with gray shading:

```
// this is C# code
class MyClass : MyBaseClass
{
```

Any new or important C# code is displayed in bold:

```
// this is C# code
class MyClass : MyBaseClass // we've already seen this bit
{
   int X; // this is interesting
```

However, any C++ code presented for comparison is presented like this, without any shading:

```
// this is C++ code
class CMyClass : public CMyBaseClass
{
```

The sample code in this appendix also takes account of the most common naming conventions when using the two languages under Windows. Hence class names in the C++ examples begin with `C`, whereas the corresponding names in the C# examples do not. Also, Hungarian notation is often used for variable names in the C++ samples only.

Terminology

You should be aware that a couple of language constructs have a different terminology in C# from that in C++. Member variables in C++ are known as *fields* in C#, and functions in C++ are known as *methods* in C#. In C#, the term *function* has a more general meaning and refers to any member of a class that contains code. This means that "function" covers methods, properties, constructors, destructors, indexers, and operator overloads. In C++, "function" and "method" are often used interchangeably in casual speech, though strictly a C++ method is a virtual member function.

If this all sounds confusing, the following table should help.

Meaning	C++ Term	C# Term
Variable that is a member of a class	Member variable	Field
Any item in a class that contains instructions	Function (or member function)	Function
Item in a class that contains instructions and is callable by name with the syntax DoSomething(/*parameters*/).	Function (or member function)	Method
Virtual function that is defined as a member of a class	Method	Virtual method

You should also be aware of the differences in terminology listed in the following table.

C++ Term	C# Term
Compound statement	Block statement
`Lvalue`	Variable expression

This appendix, where possible, uses the terminology appropriate to the language being discussed.

A Comparison of C# and C++

This section briefly summarizes the overall differences and similarities between the two languages.

Differences

The main areas in which C# differs from C++ are as follows:

- **Compile target**—C++ code usually compiles to assembly language. C# by contrast compiles to *intermediate language* (IL), which has some similarities to Java byte code. The IL is subsequently converted to native executable code by a process of Just-In-Time (JIT) compilation. The emitted IL code is stored in a file or set of files known as an assembly. An *assembly* essentially forms the unit in which IL code, along with metadata, is packaged, corresponding to a DLL or executable file that would be created by a C++ compiler.
- **Memory management**—C# is designed to free the developer from memory management bookkeeping tasks. This means that in C# you do not have to explicitly delete memory that was allocated dynamically on the heap, as you would in C++. Rather, the garbage collector periodically cleans up memory that is no longer needed. To facilitate this, C# does impose certain restrictions on how you can use variables stored on the heap and is stricter about type safety than C++.

- **Pointers** — Pointers can be used in C# just as in C++, but only in blocks of code that you have specifically marked for pointer use. For the most part, C# relies on Visual Basic/Java-style references for instances of classes, and the language has been designed in such a way that pointers are not required nearly as often as they are in C++.
- **Operator overloads** — C# does not allow you to explicitly overload as many operators as C++. This is largely because the C# compiler automates this task to some extent by using any available custom overloads of elementary operators (like =) to work out overloads of combined operators (+=) automatically.
- **Library** — Both C++ and C# rely on the presence of an extensive library. For ANSI C++ this is the standard library. C# relies on a set of classes known as the .NET base classes. The .NET base classes are based on single inheritance, whereas the standard library is based on a mixture of inheritance and templates. Also, whereas ANSI C++ keeps the library largely separate from the language itself, the interdependence in C# is much closer, and the implementation of many C# keywords is directly dependent on particular base classes.
- **Target environments** — C# is specifically designed to target programming needs in GUI-based environments (not necessarily just Windows, although the language currently only supports Windows), as well as background services such as Web services. This doesn't really affect the language itself, but is reflected in the design of the base class library. C++, by contrast, was designed for more general use in the days when command-line user interfaces were dominant. Neither C++ nor the standard library includes any support for GUI elements. On Windows, C++ developers have had to rely directly or indirectly on the Windows API for this support.
- **Preprocessor directives** — C# has some preprocessor directives, which follow the same overall syntax as in C++. But in general there are far fewer preprocessor directives in C#, because other C# language features make these less important.
- **Enumerators** — These are present in C#, but are much more versatile than their C++ equivalents, because they are syntactically fully fledged structs in their own right, supporting various properties and methods. Note that this support exists in source code only — when compiled to native executables, enumerators are still implemented as primitive numeric types, so there is no performance loss.
- **Destructors** — C# cannot guarantee when class destructors are called. In general, you should not use the programming paradigm of placing code in C# class destructors, as you can in C++, unless there are specific unmanaged resources to be cleaned up, such as file or database connections. Because the garbage collector cleans up all dynamically allocated memory, destructors are not as important in C# as they are in C++. For cases in which it is important to clean up external resources as soon as possible, C# implements an alternative mechanism involving the `IDisposable` interface.
- **Classes versus structs** — C# formalizes the difference between classes (typically used for large objects with many methods) and structs (typically used for small objects that comprise little more than collections of variables). Among other differences, classes and structs are stored differently, and structs do not support inheritance.

Similarities

Areas in which C# and C++ are very similar include the following:

- **Syntax** — The overall syntax of C# is very similar to that of C++, although numerous minor differences exist.
- **Execution flow** — C++ and C# both have roughly the same statements to control flow of execution, and these generally work in the same way in the two languages.
- **Exceptions** — Support for these in C# is essentially the same as in C++, except that C# allows finally blocks and imposes some restrictions on the type of object that can be thrown.
- **Inheritance model** — Classes are inherited in the same way in C# as in C++. Related concepts such as abstract classes and virtual functions are implemented in the same way, although there are some differences in syntax. Also, C# supports only single inheritance of classes, but multiple interface inheritance. The similarity in class hierarchy incidentally means that C# programs will normally have a very similar overall architecture to corresponding C++ programs.
- **Constructors** — Constructors work in the same way in C# as in C++, though again some differences in syntax exist.

New Features

C# introduces a number of new concepts that are not part of the ANSI C++ specification (although most of these have been introduced by Microsoft as non-standard extensions supported by the Microsoft C++ compiler). These are as follows:

- **Delegates** — C# does not support function pointers. However, a similar effect is achieved by wrapping references to methods in a special form of class known as a delegate. Delegates can be passed around between methods and used to call the methods to which they contain references, in the same way that function pointers can be in C++. What is significant about delegates is that they incorporate an object reference as well as a method reference. This means that, unlike a function pointer, a delegate contains sufficient information to call an instance method in a class.
- **Events** — Events are similar to delegates, but are specifically designed to support the callback model, in which a client notifies a server that it wants to be informed when some action takes place. C# uses events as a wrapper around Windows messages in the same way that Visual Basic does.
- **Properties** — This idea, used extensively in Visual Basic and in COM, has been imported into C#. A property is a method or get/set pair of methods in a class that have been dressed up syntactically, so to the outside world it looks like a field. Properties allow you to write code like `MyForm.Height = 400` instead of `MyForm.SetHeight(400)`.
- **Interfaces** — An interface can be thought of as an abstract class, whose purpose is to define a set of methods or properties that classes can agree to implement. The idea originated in COM. C# interfaces are not the same as COM interfaces; they are simply lists of methods and properties and such, whereas COM interfaces have other associated features such as GUIDs, but the principle is very similar. This means that C# formally recognizes the principle of interface inheritance, whereby a class inherits the definitions of functions but not any implementations.
- **Attributes** — C# allows you to decorate classes, methods, parameters, and other items in code with meta-information known as attributes. Attributes can be accessed at runtime and used to determine the actions taken by your code.

New Base Class Features

The following features are new to C# and have no counterparts in the C++ language. However, support for these features comes almost entirely from the base classes, with little or no support from the C# language syntax itself. Therefore, they are not covered in this appendix. (For more details, see Chapter 11, "Reflection," and Chapter 13, "Threading.")

- **Threading** — The C# language includes some support for thread synchronization, via the `lock` statement. (C++ has no built-in support for threads and you have to call functionality in code libraries.)
- **Reflection** — C# allows code to obtain information dynamically about the definitions of classes in compiled assemblies (libraries and executables). You can actually write a program in C# that displays information about the classes and methods that it is made up from!

There's one important feature of native C++ that is not available with C# or any other .NET language: C++ supports multiple inheritance. With C# multiple inheritance is only supported for interfaces.

The Hello World Example

Writing a Hello World application is far from original, but a direct comparison of Hello World in C++ and C# can be quite instructive for illustrating some of the differences between the two languages. In this comparison, we've tried to innovate a bit (and demonstrate more features) by displaying `"Hello World!"` both at the command line and in a message box. A slight change has also been made to the text of the message in the C++ version, in a move which we emphasize should be interpreted as a bit of fun rather than a serious statement.

The C++ version looks like this:

```
#include <iostream>
#include <Windows.h>
using namespace std;

int main(int argc, char *argv)
{
   cout << "Goodbye, World!";
   MessageBox(NULL, "Goodbye, World!", "", MB_OK);
   return 0;
}
```

Here's the C# version:

```
using System;
using System.Windows.Forms;

namespace Console1
{
```

```
    class Class1
    {
       static int Main(string[] args)
       {
          Console.WriteLine("Hello, World!");
          MessageBox.Show("Hello, World!");
          return 0;
       }
    }
}
```

Comparing the two programs tells you that the syntax of the two languages is quite similar. In particular, code blocks are marked off with braces (`{ }`), and semicolons are used as statement delimiters. Like C++, C# ignores all excess whitespace between statements. You'll go through the samples line by line, examining the features they demonstrate.

#include Statements

The C++ version of Hello World starts with a couple of preprocessor directives to include some header files:

```
#include <iostream>
#include <Windows.h>
```

These are absent from the C# version, something that illustrates an important point about the way that C# accesses libraries. In C++ you need to include header files in order for the compiler to be able to recognize the relevant symbols in your code. You need to instruct the linker separately to reference the libraries, achieved by passing command-line parameters to the linker. C# doesn't really separate compiling and linking in the way that C++ does. In C#, the command-line parameters are all that is required (and only then if you are accessing anything beyond the basic core library). By themselves, these will allow the compiler to find all the class definitions; hence explicit references in the source code are unnecessary. This is actually a much simpler way of doing it — and indeed once you've familiarized yourself with the C# model, the C++ version, in which everything needs to be referred to twice, starts to look rather strange and cumbersome.

One other point you should note is that of the two `#include` statements in the preceding C++ code, the first accesses an ANSI standard library (the `iostream` part of the standard library). The second is a Windows-specific library and is referenced to display the message box. C++ code on Windows often needs to access the Windows API because the ANSI standard doesn't have any windowing facilities. By contrast, the .NET base classes — in a sense, the C# equivalent of the ANSI standard template library — do include windowing facilities, and only the .NET base classes are used here. The C# code requires no non-standard features. (Although arguably, this point is balanced by the fact that standard C# is only available on Windows, at present.)

Although the preceding C# code happens not to have any `#include` directives, it's worth noting that some preprocessor directives (though not `#include`) are available in C# and do retain the `#` syntax.

Namespaces

The C# Hello World program starts with a namespace declaration, which is scoped by the curly braces to include the entire program. Namespaces work in exactly the same way in C# as they do in C++, providing ways to remove possible ambiguity from the names of symbols in the program. Placing items in a namespace is optional in both languages, but in C# the convention is that all items should be in a namespace. Hence, whereas it is very common to see C++ code that is not contained in a namespace, it is extremely rare to see such code in C#.

For the next part of the code, the C# and C++ versions are very similar — both use the statement `using` to indicate the namespace in which any symbols should be searched for. The only difference is a syntactical one: The statement in C# is just `using`, whereas in C++ it is `using namespace`.

Many C++ developers will be used to the old C++ library, which meant including the file `iostream.h` rather than `iostream` — in which case the `using namespace std` statement is unnecessary. The old C++ library is officially deprecated, and the previous example demonstrates how you really should be accessing the `iostream` library in C++ code.

Entry Point: Main() versus main()

The next items in the Hello World examples are the program entry points. In the C++ case this is a global function named `main()`. C# does roughly the same thing, although in C# the name is `Main()`. However, whereas in C++ `main()` is defined outside of any class, the C# version is defined as a static member of a class. This is because C# requires all functions and variables to be members of a class or struct. C# will not allow any top-level items in your program except classes and structs. To that extent C# can be regarded as enforcing stricter object-oriented practices than C++ does. Relying extensively on global and static variables and functions in C++ code tends to be regarded as poor program design anyway.

Of course, requiring that everything should be a member of a class does lead to the issue of where the program entry point should be. The answer is that the C# compiler will look for a static member method called `Main()`.This can be a member of any class in the source code, but only one class should normally have such a method. (If more than one class defines this method, a compiler switch will need to be used to indicate to the compiler which of these classes is the program entry point.) Like its C++ counterpart, `Main()` can return either a `void` or an `int`, though `int` is the more usual. Also like its C++ equivalent, `Main()` takes the same arguments — either the set of any command-line parameters passed to the program, as an array of strings, or no parameters. But as you can see from the code, strings are defined in a slightly more intuitive manner in C# than they are in C++. (In fact, the word `string` is a keyword in C#, and it maps to a class defined in the .NET base class library, `System.String`.) Also, arrays are more sophisticated in C# than in C++. Each array stores the number of elements it contains as well as the elements themselves, so there is no need to pass in the number of strings in the array separately in the C# code, as C++ does via the `argc` parameter.

Displaying the Message

Finally, you get to the lines that actually write your message — first to the console, then to a message box. In both cases these lines of code rely on calling up features from the supporting libraries for the two languages. The classes in the standard library are obviously designed very differently from those in the .NET base class library, so the details of the method calls in these code samples are different. In the C#

case, both calls are made as calls to static methods on base classes, whereas to display a message box C++ has to rely on a non-standard Windows API function, `MessageBox()`, which is not object-oriented.

The base classes are designed to be highly intuitive — arguably more so than the standard library. Without any knowledge of C#, it's immediately obvious what `Console.WriteLine()` does. If you didn't already know, you'd have a hard time figuring out what `cout <<` means. But in the commercial world of programming, being easy to understand is usually worth more than being artistic.

`MessageBox.Show()` takes fewer parameters than its C++ equivalent in this example, because it is overloaded. Other overloads that take additional parameters are available.

Also, one point that could be easy to miss is that the code demonstrates that C# uses the period, or full stop, symbol (`.`) rather than two colons (`::`) for scope resolution. `Console` and `MessageBox` are the names of classes rather than class instances! In order to access static members of classes, C# always requires the syntax `<ClassName>.<MemberName>`, whereas C++ gives you a choice between `<ClassName>::<MemberName>` and `<InstanceName>.<MemberName>` (if an instance of the class exists and is in scope).

Topic-by-Topic Comparison

The previous example provides an overview of some of the differences you'll see. The remainder of this appendix compares the two languages in detail, working systematically through the various language features of C++ and C#.

Program Architecture

This section looks in very broad terms at how the features of the two languages affect the overall architecture of programs.

Program objects

In C++ any program will consist of an entry point (in ANSI C++ this is the `main()` function, though for Windows applications this is usually named `WinMain()`), as well as various classes, structs, and global variables or functions that are defined outside of any class. Although many developers would regard good object-oriented design as meaning that as far as possible, the topmost-level items in your code are objects, C++ does not enforce this. As you've just seen, C# does enforce that idea. It lays down a more exclusively object-oriented paradigm by requiring that everything is a member of a class. In other words, the only top-level objects in your program are classes (or other items that can be regarded as special types of classes: enumerations, delegates, and interfaces). To that extent, you'll find that your C# code is forced to be even more object-oriented than would be required in C++.

File structure

In C++ the syntax by which your program is built up is very much based on the file as a unit of source code. You have, for example, source files (`.cpp` files) that contain `#include` preprocessor directives to include relevant header files. The compilation process involves compiling each source file individually, after which these object files are linked to generate the final executable. Although the final executable

does not contain any information about the original source or object files, C++ has been designed in a way that requires the developer to explicitly code around the chosen source code file structure.

With C#, the compiler takes care of the details of matching up individual source files for you. You can put your source code either in a single file or in several files, but that's immaterial for the compiler and there's no need for any file to explicitly refer to other files. In particular, there is no requirement for items to be defined before they are referenced in any individual file, as there is in C++. The compiler will happily locate the definition of each item wherever it happens to be. As a side effect of this, there isn't really any concept of linking up your own code in C#. The compiler simply compiles all your source files into an assembly (though you can specify other options such as a module — a unit that will form part of an assembly). Linking does take place in C#, but this is really confined to linking your code with any existing library code in assemblies. There is no such thing as a header file in C#.

Program entry point

In standard ANSI C++, the program entry point is by default at a function called `main()`, which normally has the signature

```
int main(int argc, char *argv[])
```

where `argc` indicates the number of arguments passed to the program, and `argv` is an array of strings giving these arguments. The first argument is always the command used to run the program itself. Windows somewhat modifies this. Windows applications traditionally start with an entry point called `WinMain()`, and DLLs start with `DllMain()`. These methods also take different sets of parameters.

In C#, the entry point follows similar principles. However, due to the requirement that all C# items are part of a class, the entry point can no longer be a global function. Instead, the requirement is that one class must have a static member method called `Main()`, as you saw earlier.

Language Syntax

C# and C++ share virtually identical syntaxes. Both languages, for example, ignore whitespace between statements, and use the semicolon to separate statements and braces to block statements together. This all means that, at first sight, programs written in either language look very much alike. However, note the following differences:

- C++ requires a semicolon after a class definition. C# does not.
- C++ permits expressions to be used as statements even if they have no effect, for example: `i+1;`.

In C# this would be flagged as an error.

Also note that, like C++, C# is case-sensitive. However, because C# is intended to be interoperable with Visual Basic (which is case-insensitive), you are strongly advised not to use names that differ only by case for any items that will be visible to code outside your project (in other words, names of public members of classes in library code). If you do use public names that differ only by case, you'll prevent Visual Basic code from being able to access your classes. (Incidentally, if you write any managed C++ code for the .NET environment, the same advice applies.)

Forward declarations

Forward declarations are neither supported nor required in C#, because the order in which items are defined in the source files is immaterial. It's perfectly fine for one item to refer to another item that is only actually defined later in that file or in a different file — as long as it is defined somewhere. This contrasts with C++, in which symbols and so on can only be referred to in a source file if they have already been declared in the same file or an included file.

No separation of definition and declaration

Something related to the lack of forward declarations in C# is that there is never any separation of declaration and definition of any item in C#. For example, in C++ it's common to write out a class something like this in the header file, where only signatures of the member functions are given, and the full definitions are specified elsewhere:

```
class CMyClass
{
public:
void MyMethod(); // definition of this function is in the C++ file,
// unless MyMethod() is inline
// etc.
```

This is not done in C#. The methods are always defined in full in the class definition:

```
class MyClass
{
   public void MyMethod()
   {
   // implementation here
```

You might at first sight think that this leads to code that is less easy to read. The beauty of the C++ way of doing it was, after all, that you could just scan through the header file to see what public functions a class exposed, without having to see the implementations of those functions. However, this facility is no longer needed in C#, partly because of modern editors (the Visual Studio editor is a folding editor, which allows you to collapse method implementations) and partly because C# has a facility to generate documentation in XML format for your code automatically.

Program Flow

Program flow is similar in C# to C++. In particular, the following statements work in exactly the same way in C# as they do in C++ and have exactly the same syntax:

- `for`
- `return`
- `goto`
- `break`
- `continue`

There are a couple of syntactical differences for the `if`, `while`, `do ... while`, and `switch` statements, and C# provides an additional control flow statement, `foreach`.

if...else

The `if` statement works in exactly the same way and has exactly the same syntax in C# as in C++, apart from one point. The condition in each `if` or `else` clause must evaluate to a `bool` type. For example, assuming `x` is an `int`, not a `bool`, the following C++-style code would generate a compilation error in C#:

```
if (x)
{
```

The correct C# syntax is

```
if (x != 0)
{
```

because the `!=` operator returns a `bool`.

This requirement is a good illustration of how the additional type safety in C# traps errors early. Runtime errors in C++ caused by writing `if (a = b)` when you meant to write `if (a == b)` are commonplace. In C# these errors are caught at compile time.

while and do while

The `while` and `do while` statements have exactly the same syntax and purpose in C# as they do in C++, except that the condition expression must evaluate to a `bool`:

```
int x;
while (x) { /* statements */ } // wrong
while (x != 0) { /* statements */ } // OK
```

switch

The `switch` statement serves the same purpose in C# as it does in C++. It is, however, more powerful in C#, because you can use a string as the test variable, something not possible in C++:

```
string myString;
// initialize myString
switch (myString)
{
   case "Hello":
      // do something
      break;
   case "Goodbye":
      // etc.
```

The syntax in C# is slightly different in that each `case` clause must explicitly exit. It is not permitted for one `case` to fall through to another `case`, unless the first `case` is empty. If you want to achieve this effect, you'll need to use the `goto` statement:

```
switch (myString)
{
   case "Hello":
      // do something;
      goto case "Goodbye"; // Will go on to execute the statements
      // in the "Goodbye" clause
   case "Goodbye":
      // do something else
      break;
   case "Black": // OK for this to fall through since it's empty
   case "White":
      // do something else // This is executed if myString contains
      // either "Black" or "White"
      break;
      default:
      int j = 3;
      break;
}
```

Microsoft has decided to enforce use of the `goto` statement in this context, in order to prevent bugs that would lead to `switch` statements falling through to the next `case` clause when the intention was actually to break.

foreach

C# provides an additional flow control statement, `foreach`. A `foreach` loop iterates through all items in an array or collection without requiring explicit specification of the indices.

A `foreach` loop on an array might look as follows. In this example, assume that `MyArray` is an array of `doubles`, and you want to output each value to the console window. To do this you would use the following code:

```
foreach (double someElement in myArray)
{
   Console.WriteLine(someElement);
}
```

Note that in this loop `someElement` is the name you will assign to the variable used to iterate through the loop — it is not a keyword.

Alternatively, you could write the preceding loop as

```
foreach (double someElement in myArray)
   Console.WriteLine(someElement);
```

because block statements in C# work in the same way as compound statements in C++.

This loop would have exactly the same effect as:

```
for (int i=0; i<myArray.Length; i++)
{
   Console.WriteLine(myArray[i]);
}
```

Note that the second version also illustrates how to obtain the number of elements in an array in C#! You learn how to declare an array in C# later in this appendix.

Unlike array element access, the `foreach` loop provides read-only access to its elements. Hence, the following code does not compile:

```
foreach (double someElement in MyArray)
someElement *= 2; // Wrong _ someElement cannot be assigned to
```

As mentioned, the `foreach` loop can be used for arrays or collections. A collection is something that has no counterpart in C++, although the concept has become common in Windows through its use in Visual Basic and COM. Essentially, a collection is a class that implements the interface `IEnumerable`. Because this involves support from the base classes, collections are explained in Chapter 9, "Collections."

Variables

Variable definitions follow basically the same pattern in C# as they do in C++:

```
int nCustomers, Result;
double distanceTravelled;
double height = 3.75;
const decimal balance = 344.56M;
```

However, as you'd expect, some of the types are different. Also, as remarked earlier, variables may only be declared locally in a method or as members of a class. C# has no equivalent to global or static (that is, scoped to a file) variables in C++. As noted earlier, variables that are members of a class are called *fields* in C#.

Note that C# also distinguishes between data types that are stored on the stack (value data types) and those that are stored on the heap (reference data types). This issue is examined in more detail shortly.

Basic data types

As with C++, C# has a number of predefined data types, and you can define your own types as classes or structs.

The data types that are predefined in C# differ somewhat from those in C++. The following table shows the types that are available in C#.

Name	Contains	Symbol
`sbyte`	Signed 8-bit integer.	
`byte`	Unsigned 8-bit integer.	
`short`	Signed 16-bit integer.	
`ushort`	Unsigned 16-bit integer.	
`int`	Signed 32-bit integer.	
`uint`	Unsigned 32-bit integer.	`U`

Name	Contains	Symbol
`long`	Signed 64-bit integer.	`L`
`ulong`	Unsigned 64-bit integer.	`UL`
`float`	Signed 32-bit floating-point value.	`F`
`double`	Signed 64-bit floating-point value.	`D`
`bool`	True or false.	
`char`	16-bit Unicode character.	`''`
`decimal`	Floating-point number with 28 significant digits.	`M`
`string`	Set of Unicode characters of variable length.	`""`
`object`	Used where you choose not to specify the type. The nearest C++ equivalent is `void*`, except that `object` is not a pointer.	

In the table, the symbol in the third column refers to the letter that can be placed after a number to indicate its type in situations for which it is desirable to indicate the type explicitly; for example, `28UL` means the number 28 stored as an unsigned long. As with C++, single quotes are used to denote characters and double quotes are used for strings. However, in C#, characters are always Unicode characters, and strings are a defined reference type, not simply an array of characters.

The data types in C# are more tightly defined than they are in C++. For example, in C++, the traditional expectation was that an `int` type would occupy 2 bytes (16 bits), but the ANSI C++ definition allowed this to be platform-dependent. Hence, on Windows, a C++ `int` occupies 4 bytes, the same as a `long`. This obviously causes quite a few compatibility problems when transferring C++ programs between platforms. On the other hand, in C# each predefined data type (except `string` and `object`, obviously!) has its total storage specified explicitly.

Because the size of each of the primitive types is fixed in C# (a primitive type is any of the types in the preceding table, except `string` and `object`), there is less need for the `sizeof` operator, though it does exist in C# but is permitted only in unsafe code (as described shortly).

Although many C# names are similar to C++ names and there is a fairly obvious intuitive mapping between many of the corresponding types, some things have changed syntactically. In particular, `signed` and `unsigned` are not recognized keywords in C#. In C++ you could use these keywords, as well as `long` and `short`, to modify other types (for example, `unsigned long, short int`). Such modifications are not permitted in C#, so the preceding table literally is the complete list of predefined data types.

Basic data types as objects

Unlike C++ (but like Java), the basic data types in C# can also be treated as objects so that you can call some methods on them. For example, in C# you can convert an integer to a string like this.

```
int i = 10;
string y = i.ToString();

// You can even write:
string y = 10.ToString();
```

The fact that you can treat the basic data types as objects reflects the close association between C# and the .NET base class library. C# actually compiles the basic data types by mapping each one onto one of the base classes, for example, `string` maps to `System.String`, `int` to `System.Int32`, and so on. So in a real sense in C#, everything is an object. However, note that this only applies for syntactical purposes. In reality, when your code is executed, these types are implemented as the underlying IL types, so there is no performance loss associated with treating basic types as objects.

All the methods available to the basic data types aren't listed here; you can find detailed information in the C# SDK documentation. You should, however, note the following:

- All types have a `ToString()` method. For the basic data types this returns a string representation of their value.
- `char` has a large number of properties that give information about its contents (`IsLetter`, `IsNumber`, and so on) as well as methods to perform conversions (`ToUpper()`, `ToLower()`).
- `string` has a very large number of methods and properties available. Some string features are shown later in this appendix.

A number of static member methods and properties are also available:

- Integer types have `MinValue` and `MaxValue` to indicate the minimum and maximum values that may be contained in the type.
- The `float` and `double` types also have a property, `Epsilon`, which indicates the smallest possible value greater than zero that may be stored.
- Separate values, `NaN` (not a number; that is, undefined), `PositiveInfinity`, and `NegativeInfinity` are defined for `float` and `double`. Results of computations will return these values as appropriate (for example, dividing a positive number by zero returns `PositiveInfinity`, whereas dividing zero by zero returns `NaN`). These values are available as static properties.
- Many types, including all the numeric types, have a static `Parse()` method that allows you to convert from a string: `double D = double.Parse("20.5");`.

Note that static methods in C# are called by specifying the name of the type: `int.MaxValue` and `float.Epsilon`.

Casting between the basic data types

Casting is the process of converting a value stored in a variable of one data type to a value of another data type. In C++ this can be done either implicitly or explicitly:

```
float f1 = 40.0;
long l1 = f1; // implicit
short s1 = (short) l1; // explicit, old C style
short s2 = short (f1); // explicit, new C++ style
```

If the cast is specified explicitly, this means that you have explicitly indicated the name of the destination data type in your code. C++ allows you to write explicit casts in either of two styles — the old C style, in which the name of the data type was enclosed in brackets, or the new style, in which the name of the variable is enclosed in brackets. Both styles are demonstrated in the preceding example and are syntactical preferences — the choice of style has no effect on the code. In C++ it is legal to convert between any of the basic data types. However, if there is a risk of a loss of data because the destination data type has a smaller range than the source data type, the compiler may issue a warning, depending on your warning level settings. In the preceding example, the implicit cast may cause loss of data, which means it will normally cause the compiler to issue a warning. Explicitly specifying the conversion is really a way of telling the compiler that you know what you are doing — as a result this will normally suppress any warnings.

Because C# is designed to be more type-safe than C++, it is less flexible about converting between the data types. It also formalizes the notion of explicit and implicit casts. Certain conversions are defined as implicit casts, meaning that you are allowed to perform them using either the implicit or the explicit syntax. Other conversions can only be done using explicit casts, which means the compiler will generate an error (not a warning, as in C++!) if you try to carry out the cast implicitly.

The rules in C# concerning which of the basic numeric data types can be converted to which other types are quite logical. Implicit casts are the ones that involve no risk of loss of data — for example, `int` to `long` or `float` to `double`. Explicit casts might involve data loss, due to an overflow error, sign error, or loss of the fractional part of a number (for example, `float` to `int`, `int` to `uint`, or `short` to `ulong`). In addition, because `char` is considered somewhat distinct from the other integer types, converting to or from a `char` can only be done explicitly.

For example, the following lines all count as valid C# code:

```
float f1 = 40.0F;
long l1 = (long)f1; // explicit due to possible rounding error
short s1 = (short) l1; // explicit due to possible overflow error
int i1 = s1; // implicit _ no problems
uint i2 = (uint)i1; // explicit due to possible sign error
```

Note that in C#, explicit casts are always done using the old C-style syntax. The new C++ syntax cannot be used:

```
uint i2 = uint(i1); // wrong syntax _ this won't compile
```

Overflow checking

C# offers the ability to perform arithmetic operations in a checked context. This means that the .NET runtime detects any overflows and throws an exception (specifically an `OverFlowException`) if an overflow does occur. This feature has no counterpart in C++:

```
checked
{
   int i1 = -3;
   uint i2 = (uint)i1;
}
```

Because of the checked context, the second line will throw an exception. If you had not specified `checked`, no exception would be thrown and the variable `i2` would contain garbage.

With the `unchecked` keyword, overflow checking does not happen. The default behavior of overflow checking is set with the csc compiler option `/checked`.

Strings

String handling is far easier in C# than it ever was in C++. This is because of the existence of `string` as a basic data type that is recognized by the C# compiler. There is no need to treat strings as arrays of characters in C#.

The closest equivalent to C#'s `string` data type in C++ is the `string` class in the standard library. However, the C# `string` differs from the C++ `string` in the following main ways.

- The C# `string` contains Unicode, not ANSI, characters.
- The C# `string` has many more methods and properties than the C++ version does.
- In C++ the standard library `string` class is no more than a class supplied by the library, whereas in C# the language syntax specifically supports the `string` class as part of the language.

Escape sequences

C# uses the same method of escaping special characters as C++ — a backslash. The following table provides a complete list of escape sequences.

Escape Sequence	Character Name	Unicode Encoding
`\'`	Single quote	0x0027
`\"`	Double quote	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\a`	Alert	0x0007
`\b`	Backspace	0x0008
`\f`	Form feed	0x000C
`\n`	Newline	0x000A
`\r`	Carriage return	0x000D
`\t`	Horizontal tab	0x0009
`\v`	Vertical tab	0x000B

This basically means that the codes used in C# are the same as those used in C++, except that C# doesn't recognize `\?`.

A couple of differences exist between escape characters in C++ and C#:

- The escape sequence `\0` is recognized in C#. However, it is not used as string terminator in C# and so can be embedded in strings. C# strings work by separately storing their lengths so no character is used as a terminator. Hence C# strings really can contain any Unicode character.
- C# has an additional escape sequence, `\uxxxx` (or equivalently `\Uxxxx`), where `xxxx` represents a 4-digit hexadecimal number. `\uxxxx` represents the Unicode character `xxxx`; for example, `\u0065` represents `'e'`. However, unlike the other escape sequences, `\uxxxx` can be used in variable names as well as in character and string constants. For example, the following is valid C# code:

```
int r\u0065sult; // has the same effect as int result;
result = 10;
```

C# also has an alternative method for expressing strings that is more convenient for strings that contain special characters. Placing an `@` symbol in front of the string prevents any characters from being escaped. These strings are known as verbatim strings. For example, to represent the string `C:\Book\Chapter2`, you could write either `"C:\\Book\\Chapter2"` or `@"C:\Book\Chapter2"`. Interestingly, this also means you can include carriage returns in verbatim strings without escaping them:

```
string Message = @"This goes on the first line
and this goes on the next line";
```

Value types and reference types

C# divides all data types into two types: *value* types and *reference* types. This distinction has no equivalent in C++, where variables always implicitly contain values, unless a variable is specifically declared as a reference to another variable.

In C#, a value type actually contains its value. All the predefined data types in C# are value types, except for `object` and `string`. If you define your own structs or enumerations, these will also be value types. This means that the simple data types in C# generally work in exactly the same way as in C++ when you assign values to them:

```
int i = 10;
long j = i; // creates another copy of the value 10
i = 15; // has no effect on j
```

A reference type, as its name implies, contains only a reference to where the data is kept in memory. Syntactically, this works the same way as references in C++, but in terms of what is actually happening, C# references are closer to C++ pointers. In C#, `object` and `string` are reference types, as are any classes that you define yourself. C# references can be reassigned to point to different data items, in much the same way that C++ pointers can. Also, C# references can be assigned the value `null` to indicate that they don't refer to anything. For example, suppose you have a class called `MyClass`, which has a public property, `Width`:

```
MyClass My1 = new MyClass(); // In C#, new simply calls a constructor.
My1.Width = 20;
MyClass My2 = My1; // My2 now points to the same memory
// location as My1.
My2.Width = 30; // Now My1.Width = 30 too because My1 and My2
// point to the same location.
My2 = null; // Now My2 doesn't refer to anything.
// My1 still refers to the same object.
```

It is not possible in C# to declare a particular variable programmatically as a value or as a reference type — that is determined exclusively by the data type of the variable.

Value and reference types have implications for memory management, because reference types are always stored on the heap, whereas value types are usually on the stack, unless they are fields in a reference object, in which case they will reside on the heap. This is covered in more detail in the next section, "Memory Management."

Initialization of variables

In C++ variables are never initialized unless you explicitly initialize them (or in the case of classes, supply constructors). If you don't, the variables will contain whatever random data happened to be in the memory location at the time — this reflects the emphasis on performance in C++. C# puts more emphasis on avoiding runtime bugs, and is therefore stricter about initializing variables. The rules in C# are as follows:

- Variables that are member fields are by default initialized by being zeroed out if you do not explicitly initialize them. This means that numeric value types will contain zero, `bool`s will contain `false`, and all reference types (including `string` and `object`) will contain the `null` reference). Structs will have each of their members zeroed out.
- Variables that are local to methods are not initialized by default. However, the compiler will raise an error if a local variable is used before it is initialized. You can initialize a variable by calling its default constructor (which zeros out the memory):

```
// variables that are local to a method
int x1; // At this point x1 contains random data
//int y = x1; // This commented out line would produce a compilation error
// as x1 is used before it is initialized
x1 = new int(); // Now x1 will contain zero and is initialized
```

Boxing

In some cases, you might want to treat a value type as if it were a reference type. This is achieved by a process known as *boxing*. Syntactically, this just means casting the variable to an object:

```
int j = 10;
object boxedJ = j;  // boxing
```

Boxing acts like any other cast, but you should be aware that it means that the contents of the variable will be copied to the heap and a reference created (because the object `boxedJ` is a reference type).

The common reason for boxing a value is to pass it to a method that expects a reference type as a parameter. You can also unbox a boxed value simply by casting it back to its original type:

```
int j = 10;
object boxedJ = j; // boxing
int k = (int) boxedJ; // unboxing
```

Note that the process of unboxing raises an exception if you attempt to cast to the wrong type and no cast is available for you to do the conversion.

Memory Management

In C++, variables (including instances of classes or structs) may be stored on the stack or the heap. In general, a variable is stored on the heap if it, or some containing class, has been allocated with `new`, and it is placed on the stack otherwise. This means that through your choice of whether to allocate memory for a variable dynamically using `new`, you have complete freedom to choose whether a variable should be stored on the stack or the heap. (But obviously, due to the way the stack works, data stored on the stack will only exist as long as the corresponding variable is in scope.)

C# works very differently in this regard. One way to understand the situation in C# is by thinking of two common scenarios in C++. Look at these two C++ variable declarations:

```
int j = 30;
CMyClass *pMine = new CMyClass;
```

Here the contents of `j` are stored on the stack. This is exactly the situation that exists with C# value types. The `MyClass` instance is, however, stored on the heap, and a pointer to it is on the stack. This is basically the situation with C# reference types, except that in C# the syntax dresses the pointer up as a reference. The equivalent in C# is:

```
int J = 30;
MyClass Mine = new MyClass();
```

This code has pretty much the same effect in terms of where the objects are stored as does the previous C++ code — the difference is that `MyClass` is syntactically treated as a reference rather than a pointer.

The big difference between C++ and C# is that C# does not allow you to choose how to allocate memory for a particular instance. For example, in C++ you could if you wanted do this:

```
int* pj = new int(30);
CMyClass Mine;
```

This will cause the `int` type to be allocated on the heap, and the `CMyClass` instance to be allocated on the stack. You cannot do this in C# because in C#, an `int` is a value type, whereas any class is always a reference type.

The other difference is that there is no equivalent to the C++ `delete` operator in C#. Instead, with C# the .NET garbage collector periodically comes in and scans through the references in your code in order to identify which areas of the heap are currently in use by your program. It is then automatically able to remove all the objects that are no longer in use. This technique effectively saves you from having to free up any memory yourself on the heap.

To summarize, in C# the following are always value types:

- All simple predefined types (except `object` and `string`)
- All structs
- All enumerations

The following are always reference types:

- `object`
- `string`
- All classes

The new operator

The `new` operator has a very different meaning in C# compared to C++. In C++, `new` indicates a request for memory on the heap. In C#, `new` simply means that you are calling the constructor of a variable. However, the action is similar to the extent that if the variable is a reference type, calling its constructor will implicitly allocate memory for it on the heap. For example, suppose you have a class, `MyClass`, and a struct, `MyStruct`. In accordance with the rules of C#, `MyClass` instances will always be stored on the heap and `MyStruct` instances on the stack:

```
MyClass Mine; // Just declares a reference. Similar to declaring
// an uninitialized pointer in C++.

Mine = new MyClass(); // Creates an instance of MyClass. Calls no-
// parameter constructor. In the process, allocates
// memory on the heap.

MyStruct Struct; // Creates a MyStruct instance but does not call
// any constructor. Fields in MyStruct will be
// uninitialized.

Struct = new MyStruct(); // Calls constructor, so initializing fields.
// But doesn't allocate any memory because Struct
// already exists on stack.
```

It is possible to use `new` to call the constructor for predefined data types, too:

```
int x = new int();
```

This has the same effect as:

```
int x = 0;
```

Note that this is not the same as:

```
int x;
```

This latter statement leaves `x` uninitialized (if `x` is a local variable).

The delete operator

With C++ the `delete` operator releases memory that was allocated with the `new` operator. With objects that are deleted with the `delete` operator, the destructor of the class is invoked.

C# doesn't have a delete operator because here memory is automatically freed by the garbage collector. However, there are still destructors that have a different meaning to the C++ destructor, and ways to release unmanaged memory in a defined way as is discussed later with destructors.

Methods

Methods in C# are defined in the same way as functions in C++, apart from the fact that C# methods must always be members of a class, and the definition and declaration are always merged in C#:

```
class MyClass
{
   public int MyMethod()
   {
      // implementation
```

One restriction, however, is that member methods may not be declared as `const` in C#. The C++ facility for methods to be explicitly declared as `const` (in other words, not modifying their containing class instance) looked originally like a good compile-time check for bugs, but tended to cause problems in practice. This was because it's common for methods that do not alter the public state of the class to alter the values of private member variables (for example, for variables that are set on first access). It's not uncommon in C++ code to use the `const_cast` operator to circumvent a method that has been declared as `const`. In view of these problems, Microsoft decided not to allow `const` methods in C#.

Method parameters

As in C++, parameters are by default passed to methods by value. If you want to modify this behavior, you can use the keywords `ref` to indicate that a parameter is passed by reference, and `out` to indicate that it is an output parameter (always passed by reference). If you do this, you need to indicate the fact both in the method definition and when the method is called:

```
public void MultiplyByTwo(ref double d, out double square)
{
   d *= 2;
   square = d * d;
}

// Later on, when calling method:
double value, square;
value = 4.0;
MultiplyByTwo(ref value, out square);
```

Passing by reference means that the method can modify the value of the parameter. You might also pass by reference to improve performance when passing large structs, because, just as in C++, passing by reference means that only the address is copied. Note, however, that if you are passing by reference for performance reasons, the called method will still be able to modify the value of the parameter — C# does not permit the `const` modifier to be attached to parameters in the way that C++ does.

Output parameters work in much the same way as reference parameters, except that they are intended for cases in which the called method supplies the value of the parameter rather than modifying it. Hence the requirements when a parameter is initialized are different. C# requires that a `ref` parameter is initialized before being passed to a method but requires that an `out` parameter is initialized within the called method before being used.

Method overloads

Methods may be overloaded in the same way as in C++. However, C# does not permit default parameters to methods. This must be simulated with overloads:

```
// In C++, you can do this:
double DoSomething(int someData, bool Condition = true)
{
   // etc.
```

Whereas in C#, you have to do this:

```
double DoSomething(int someData)
{
   DoSomething(someData, true);
}

double DoSomething(int someData, bool condition)
{
   // etc.
```

Properties

Properties have no equivalent in ANSI C++, though they have been introduced as extensions in Microsoft Visual C++. A property is a method or pair of methods that are dressed syntactically to appear to calling code as if they were a field. They exist for the situation in which it is more intuitive for a method to be called with the syntax of a field — an obvious example is the case of a private field that is to be encapsulated by being wrapped by public accessor methods. Suppose a class has such a field, `length`, of type `int`. In C++ you would encapsulate it with methods `GetLength()` and `SetLength()`, and you would need to access it from outside the class like this:

```
// MyObject is an instance of the class in question
MyObject.SetLength(10);
int length = MyObject.GetLength();
```

In C# you could implement these methods instead as `get` and `set` accessors of a property named `Length`. Then you could write:

```
// MyObject is an instance of the class in question
MyObject.Length = 10;
int Length = MyObject.Length;
```

To define these accessors, you would define the property like this:

```
class MyClass
{
   private int length;
   public int Length
   {
      get
      {
         return length;
      }
      set
      {
         length = value;
      }
   }
}
```

Although here you have implemented the `get` and `set` accessors to simply return or set the length field, you can put any other C# code you want in these accessors, just as you could for a method. For example, you might add some data validation to the set accessor. Note that the set accessor returns `void` and takes an extra implicit parameter, which has the name `value`.

It is possible to omit either the `get` or `set` accessor from the property definition, in which case the corresponding property respectively becomes either write-only or read-only.

Operators

The meanings and syntaxes of operators is much the same in C# as in C++. The following operators by default have the same meaning and syntax in C# as in C++:

- The binary arithmetic operators `+`, `-`, `*`, `/`, `%`
- The corresponding arithmetic assignment operators `+=`, `-=`, `*=`, `/=`, `%=`
- The unary operators `++` and `--` (both prefix and postfix versions)
- The comparison operators `!=`, `==`, `<`, `<=`, `>`, `>=`
- The shift operators `>>` and `<<`
- The logical operators `&`, `|`, `&&`, `||`, `~`, `^`, `!`
- The assignment operators corresponding to the logical operators: `>>=`, `<<=`, `&=`, `|=`, `^=`
- The ternary (conditional) operator `? :`

The symbols `()`, `[]`, and `,` (comma) also have broadly the same effect in C# as they do in C++.

You'll need to be careful of the following operators because they work differently in C# from in C++:

- Assignment (`=`)
- `new`
- `this`

Scope resolution in C# is represented by `.`, not by `::` (`::` has no meaning in C#). Also, the `delete` and `delete[]` operators do not exist in C#. They are not necessary because the garbage collector automatically handles cleaning up of memory on the heap. However, C# also supplies three other operators that do not exist in C++: `is`, `as`, and `typeof`. These operators are related to obtaining type information for an object or class.

Assignment operator (=)

For simple data types, `=` simply copies the data. However, when you define your own classes, C++ regards it as largely the responsibility of the developer to indicate the meaning of `=` for your classes. By default in C++, `=` causes a shallow memberwise copy of any variable, class, or struct to be made. However, programmers overload this operator to carry out more complex assignment operations.

In C#, the rules governing what the assignment operator means are much simpler; it also does not permit you to overload `=` at all — its meaning is defined implicitly in all situations.

The situation in C# is as follows:

- For simple data types, = simply copies the values as in C++.
- For structs, = does a shallow copy of the struct — a direct memory copy of the data in the struct instance. This is similar to its behavior in C++.
- For classes, = copies the reference; that is, the address and not the object. This is *not* the behavior in C++.

If you want to be able to copy instances of classes, the usual way in C# is to override a method, `MemberwiseCopy()`, which all classes in C# by default inherit from the class `System.Object`, the grandfather class from which all C# classes implicitly derive.

this

The `this` operator has the same meaning as in C++, but it is a reference rather than a pointer. For example, in C++ you can do this:

```
this->m_MyField = 10;
```

However, in C#, you must do this:

```
this.MyField = 10;
```

`this` is used in the same way in C# as in C++. For example, you can pass it as a parameter in method calls, or use it to make it explicit that you are accessing a member field of a class. In C#, there are a couple of other situations that syntactically require use of `this`, which are mentioned in the section on classes.

new

As mentioned earlier, the `new` operator has a very different meaning in C#, being interpreted as a constructor, to the extent that it forces an object to initialize, rather than as a request for dynamic memory allocation.

Classes and Structs

In C++, classes and structs are extremely similar. Formally, the only difference is that members of a struct are by default public, whereas members of a class are by default private. In practice, however, many programmers prefer to use structs and classes in different ways, reserving use of structs for data objects, which contain only member variables (in other words, no member functions or explicit constructors).

C# reflects this traditional difference of usage. In C# a class is a very different type of object from a struct, so you'll need to consider carefully whether a given object is best defined as a class or as a struct. The most important differences between C# classes and C# structs are:

- Structs do not support inheritance, other than the fact that they derive from `System.ValueType`. It is not possible to inherit from a struct, nor can a struct inherit from another struct or class.
- Structs are value types. Classes are always reference types.

- Structs allow you to organize the way that fields are laid out in memory, and to define the equivalent of C++ unions.
- The default (no-parameter) constructor of a struct is always supplied by the compiler and cannot be replaced.

Because classes and structs are so different in C#, they are treated separately in this appendix.

Classes

Classes in C# by and large follow the same principles as in C++, although there are a few differences in both features and syntax. This section goes over the differences between C++ classes and C# classes.

Definition of a class

Classes are defined in C# using what at first sight looks like much the same syntax as in C++:

```
class MyClass : MyBaseClass
{
   private string SomeField;
   public int SomeMethod()
   {
      return 2;
   }
}
```

Behind this initial similarity, numerous differences exist in the detail:

- There is no access modifier on the name of the base class. Inheritance is always public.
- A class can only be derived from one base class (although it might also be derived from any number of interfaces). If no base class is explicitly specified, the class will automatically be derived from `System.Object`, which will give the class all the functionality of `System.Object`, the most commonly used of which is `ToString()`.
- Each member is explicitly declared with an access modifier. There is no equivalent to the C++ syntax in which one access modifier can be applied to several members:

```
public: // you can't use this syntax in C#
   int MyMethod();
   int MyOtherMethod();
```

- Methods cannot be declared as `inline`. This is because C# is compiled to IL. Any inlining happens at the second stage of compilation — when the Just-In-Time (JIT) compiler converts from IL to native machine code. The JIT compiler has access to all the information in the IL to determine which methods can suitably be inlined without any need for guidance from the developer in the source code.
- The implementation of methods is always placed with the definition. There is no ability to write the implementation outside the class, as C++ allows.
- Whereas in ANSI C++, the only types of class member are variables, functions, constructors, destructors, and operator overloads, C# also permits delegates, events, and properties.

- The access modifiers `public`, `private`, and `protected` have the same meaning as in C++, but two additional access modifiers are available:
  - `internal` restricts access to other code within the same assembly.
  - `protected internal` restricts access to derived classes that are within the same assembly.
- Initialization of variables is permitted in the class definition in C#.
- C++ requires a semicolon after the closing brace at the end of a class definition. This is not required in C#.

Initialization of member fields

The syntax used to initialize member fields in C# is very different from that in C++, although the end effect is identical.

Instance members

In C++, instance member fields are usually initialized in the constructor initialization list:

```
MyClass::MyClass()
: m_MyField(6)
{
// etc.
```

In C# this syntax is wrong. The only items that can be placed in the constructor initializer (which is the C# equivalent of the C++ constructor initialization list) is another constructor (a constructor of the same class or a constructor of the base class). Instead, the initialized value is marked with the definition of the member in the class definition:

```
class MyClass
{
   private int MyField = 6;
```

Note that in C++, this would be an error because C++ uses roughly this syntax to define pure virtual functions. In C# this is fine, because C# does not use the `=0` syntax for this purpose (it uses the `abstract` keyword instead).

Static fields

In C++ static fields are initialized via a separate definition outside the class:

```
int MyClass::MyStaticField = 6;
```

Indeed in C++, even if you do not want to initialize a static field, you must include this statement in order to avoid a link error. By contrast, C# does not expect such a statement, because variables are only declared within the class declaration in C#:

```
class MyClass
{
   private static int MyStaticField = 6;
```

Constructors

The syntax for declaring constructors in C# is the same as that for inline constructors defined in the class definition in C++:

```
class MyClass
{
   public MyClass()
   {
      // construction code
   }
```

As with C++, you can define as many constructors as you want, provided they take different numbers or types of parameters. (Note that, as with methods, default parameters are not permitted — you must simulate this with multiple overloads.)

For derived classes in a hierarchy, constructors work in C# in basically the same way as in C++. By default, the constructor at the top of the hierarchy (this is always `System.Object`) is executed first, followed in order by constructors down the tree.

Static constructors

C# also allows the concept of a static constructor, which is executed only once, and can be used to initialize static variables. The concept has no direct equivalent in C++:

```
class MyClass
{
   static MyClass()
   {
      // static construction code
   }
```

Static constructors are very useful in that they allow static fields to be initialized with values that are determined at runtime (for example, they can be set to values that are read in from a database). This kind of effect is possible in C++ but takes a fair amount of work and results in a fairly messy-looking solution. The most common way would be to have a function that accesses the static member variable, and implement the function so that it sets the value of the variable the first time it is called.

Note that a static constructor has no access specifier — it is not declared as public, private, or anything else. An access specifier would be meaningless because the static constructor is only ever called by the .NET runtime when the class definition is loaded. It cannot be called by any other C# code.

C# does not specify exactly when a static constructor will be executed, except that it will be after any static fields have been initialized but before any objects of the class are instantiated or static methods on the class are actually used.

Default constructors

As in C++, C# classes typically have a no-parameter default constructor, which simply calls the no-parameter constructor of the immediate base class and then initializes all fields to their default parameters. Also as in C++, the compiler will generate this default constructor only if you have not supplied

any constructors explicitly in your code. If any constructors are present in the class definition, whether or not a no-parameter constructor is included, then these constructors will be the only ones available.

As in C++ it is possible to prevent instantiation of a class by declaring a private constructor as the only constructor:

```
class MyClass
{
   private MyClass()
   {
   }
```

This also prevents instantiation of any derived classes; however with C# it is also possible to use the keyword `sealed` to prevent derived classes.

Constructor initialization lists

C# constructors might have something that looks like a C++ constructor initialization list. However, in C# this list can only contain at most one member and is known as a *constructor initializer*. The item in the initializer must either be a constructor of the immediate base class, or another constructor of the same class. The syntax for these two options uses the keywords `base` and `this`, respectively:

```
class MyClass : MyBaseClass
{
   MyClass(int X)
   : base(X) // executes the MyBaseClass 1-parameter constructor
   {
      // other initialization here
   }

   MyClass()
   : this (10) // executes the 1-parameter MyClass constructor
   // passing in the value of 10
   {
      // other initialization here
   }
```

If you do not explicitly supply any constructor initialization list, the compiler will implicitly supply one that consists of the item `base()`. In other words, the default initializer calls the default base class constructor. This behavior mirrors that of C++.

Unlike C++, you cannot place member variables in a constructor initialization list. However, that is just a matter of syntax — the C# equivalent is to mark their initial values in the class definition. A more serious difference is the fact that you can only place one other constructor in the list. This will affect the way you plan out your constructors, though this is arguably beneficial because it forces you into a well-defined and effective paradigm for arranging your constructors. This paradigm is indicated in the preceding code: the constructors all follow a single path for the order in which various constructors are executed.

Destructors

C# implements a very different programming model for destructors compared to C++. This is because the garbage collection mechanism in C# implies that

- There is less need for destructors, because dynamically allocated memory is removed automatically.
- Because it is not possible to predict when the garbage collector will actually destroy a given object, if you do supply a destructor for a class, it is not possible to predict precisely when that destructor is executed.

Because memory is cleaned up behind the scenes in C#, you will find that only a small portion of your classes actually requires destructors. For those that do, (these will be classes that maintain external unmanaged resources such as file and database connections), C# has a two-stage destruction mechanism:

1. The class should derive from the `IDisposable` interface, and implement the method `Dispose()`. Client code should explicitly call this method to indicate it has finished with an object and needs to clean up resources. (Interfaces are covered later in this appendix.)
2. The class should separately implement a destructor, which is viewed as a reserve mechanism, in case a client does not call `Dispose()`.

The usual implementation of `Dispose()` looks like this:

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
   if (disposing)
   {
      // Cleanup of managed resources here
   }
   // Cleanup of unmanaged resources
}
```

`System.GC` is a base class that represents the garbage collector. `SuppressFinalize()` is a method that informs the garbage collector that there is no need to call the destructor for the object that it is destroying. Calling `SuppressFinalize()` is important, because there is a performance hit if the object has a destructor that needs to be called while the garbage collector is doing its job; the consequence of this is that the actual freeing of that object's managed memory resources will be considerably delayed.

The syntax for the actual destructor is basically the same in C# as in C++. Note that in C# there is no need to declare the destructor as virtual — the compiler will assume it is. You should also not supply an access modifier:

```
class MyClass
{
   ~MyClass()
   {
      // clean up resources
   }
```

Although the `Dispose()` method is normally called explicitly by clients, C# does allow an alternative syntax that ensures that the compiler will arrange for it to be called. If the variable is declared inside a `using()` block, it will be scoped to the `using` block and its `Dispose()` method will be called on exiting the block:

```
using (MyClass MyObject = new MyClass())
{
   // code
} // MyObject.Dispose() will be implicitly called on leaving this block
```

Note that this code will only compile successfully if `MyClass` implements the interface `IDisposable` with the method `Dispose()`. If you don't want to use the `using` syntax, you can omit either or both of the two steps involved in the destructor sequence (implementing `Dispose()` and implementing a destructor), but normally you would implement both steps.

Inheritance

Inheritance works in basically the same way in C# as in C++, with the exception that multiple implementation inheritance is not supported. Microsoft believes that multiple inheritance leads to code that is less well structured and harder to maintain, and so a decision has been made to omit this feature from C#.

```
class MyClass : MyBaseClass
{
   // etc.
```

In C++, a pointer to a class can also point to an instance of a derived class. (Virtual functions do, after all, depend on this fact!) In C#, classes are accessed via references, but the equivalent rule holds. A reference to a class can refer to instances of that class or to instances of any derived class:

```
MyBaseClass Mine;
Mine = new MyClass(); //OK if MyClass is derived from MyBaseClass
```

If you want a reference to be able to refer to anything (the equivalent of `void*` in C++), you can define it as `object` in C#, because C# maps `object` to the `System.Object` class (from which all other classes are derived):

```
object Mine2 = new MyClass();
```

Virtual and non-virtual functions

Virtual functions are supported in C# in the same way as in C++. However, some syntactical differences in C# are designed to eliminate certain potential ambiguities in C++. This means that certain types of errors, which only appear at runtime in C++, will be identified at compile time in C#.

Also note that in C#, classes are always accessed through a reference (equivalent to access through a pointer in C++).

In C++, if you require a function to be virtual, all you need to do is to specify the `virtual` keyword in both the base and derived class. By contrast, in C# you need to declare the function as `virtual` in the base class and as `override` in any derived class versions:

```
class MyBaseClass
{
   public virtual void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}

class MyClass : MyBaseClass
{
   public override void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}
```

The point of this syntax is that it makes it explicit to the compiler how you want your function to be interpreted, and it means that there is no risk of any bugs where, for example, you type in a slightly incorrect method signature in an override version and therefore end up defining a new function when you intended to override an existing one. The compiler will flag an error if a function is marked as an override and the compiler cannot identify a version of it in any base class.

If the function is not virtual, you can still define versions of that method in the derived class, in which case the derived class version is said to hide the base class version. In this case, which method gets called depends solely on the type of the reference used to access the class, just as it depends on the pointer type used to access a class in C++.

Using C#, a method from the base class can be hidden by using the `new` keyword in the derived class:

```
class MyBaseClass
{
   public void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}

class MyClass : MyBaseClass
{
   new public void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}
```

If you do not mark the new version of the class explicitly as `new`, the code will still compile but the compiler will flag a warning. This warning is intended to guard against any subtle runtime bugs caused by, for example, writing a new base class, in which a method has been added that happens to have the same name as an existing method in the derived class.

You can declare abstract functions in C# just as you can in C++ (in C++ these are also termed pure virtual functions). The syntax, however, is different in C#: instead of using `=0` at the end of the definition you use the keyword `abstract`:

```
C++:

public:
   virtual void DoSomething(int X) = 0;

C#:

public abstract void DoSomething(int X);
```

As in C++, you can only instantiate a class if it contains no abstract methods itself, and if it provides implementations of any abstract methods that have been defined in any of its base classes.

Structs

The syntax for defining structs in C# follows that for defining classes:

```
struct MyStruct
{
   private SomeField;

   public int SomeMethod()
   {
      return 2;
   }
}
```

Inheritance and the associated concepts virtual and abstract functions are not permitted. Otherwise, the basic syntax is identical with classes except that the keyword `struct` replaces `class` in the definition.

However, a couple of differences exist between structs and classes when it comes to construction. In particular, structs always have a default constructor that zeros out all the fields, and this constructor is still present even if you define other constructors of your own. Also, it is not possible to define a no-parameter constructor explicitly to replace the default one. You can only define constructors that take parameters. In this respect, structs in C# differ from their C++ counterparts.

Unlike classes in C#, structs are value types. This means that a statement such as

```
MyStruct Mine;
```

actually creates an instance of `MyStruct` on the stack, just as the same statement would do in C++.

However, in C#, this instance is uninitialized unless you explicitly call the constructor:

```
MyStruct Mine = new MyStruct();
```

If the member fields of `MyStruct` are all public, you can alternatively initialize it by initializing each member field separately.

Constants

The C++ keyword `const` has quite a large variety of uses. For example, you can declare variables as `const`, which indicates that their values are usually set at compile time and cannot be modified by any assignment statement at runtime (although there is a tiny bit of flexibility because the value of a `const` member variable can be set in a constructor initialization list, which implies that in this case the value can be calculated at runtime). You can also apply `const` to pointers and references to prevent those pointers or references from being used to modify the data to which they point, and you can also use the `const` keyword to modify the definitions of parameters passed to functions. Here, `const` indicates that a variable that has been passed by reference or via a pointer should not be modified by the function. Also, as mentioned earlier, member functions themselves can be declared as `const` to indicate that they do not change their containing class instance.

C# also allows use of the `const` keyword to indicate that a variable cannot be changed. However, use of `const` is far more restricted in C# than in C++. In C#, the *only* use of `const` is to fix the value of a variable (or of the referent of a reference) at compile time. It cannot be applied to methods or parameters. On the other hand, C# is more flexible than C++, to the extent that the syntax in C# does allow a little more flexibility for initializing `const` fields at runtime than C++ does.

The syntax for declaring constants is very different in C# than in C++, so it's discussed in some detail. The C# syntax makes use of two keywords, `const` and `readonly`. The `const` keyword implies that a value is set at compile time, whereas `readonly` implies it is set once at runtime, in a constructor.

Because everything in C# must be a member of a class or struct, there is of course no direct equivalent in C# to global constants in C++. This functionality must be obtained using either enumerations or static member fields of a class.

Constants that are associated with a class (static constants)

The usual way of defining a static constant in C++ is as a `static const` member of a class. C# approaches this in broadly the same way but with a simpler syntax:

```
C++ syntax:

int CMyClass :: MyConstant = 2;
class CMyClass
{
   public:
   static const int MyConstant;

C# syntax:

class MyClass
{
   public const int MyConstant = 2;
```

Note that in C# you do not explicitly declare the constant as static — doing so would result in a compilation error. It is, of course, implicitly static, because there is no point storing a constant value more than once, and hence it must always be accessed as a static field:

```
int SomeVariable = MyClass.MyConstant;
```

Things get a bit more interesting when you want your static constant to be initialized with some value that is calculated at runtime. C++ simply has no facility to allow this. If you want to achieve that effect, you will have to find some means of initializing the variable the first time it is accessed, which means you will not be able to declare it as `const` in the first place. Here C# scores easily over C++, because static constants initialized at runtime are easy to define in C#. You define the field as `readonly`, and initialize it in the static constructor:

```
class MyClass
{
   public static readonly int MyConstant;
   static MyClass()
   {
      // work out and assign the initial value of MyConstant here
   }
```

Instance constants

Constants associated with class instances are always initialized with values calculated at runtime. (If their values were calculated at compile time that would, by definition, make them static.)

In C++, such constants must be initialized in the initialization list of a class constructor. This, to some extent, restricts your flexibility in calculating the values of these constants because the initial value must be something that you can write down as an expression in the constructor initialization list:

```
class CMyClass
{
   public:
   const int MyConstInst;
   CMyClass()
   : MyConstInst(45)
   {
```

In C# the principle is similar, but the constant is declared as `readonly` rather than `const`. This means that its value is set in the body of the constructor giving you a bit more flexibility, because you can use any C# statements in the process of calculating its initial value. (Recall that you cannot set the values of variables in constructor initializers in C# — you can only call one other constructor.)

```
class MyClass
{
   public readonly int MyConstInst;
   MyClass()
   {
      // work out and initialize MyConstInst here
```

In C#, if a field is declared as `readonly`, it can only be assigned to in a constructor.

Operator Overloading

Operator overloading in C# also shares some similarities with C++. The key difference is that C++ allows the vast majority of its operators to be overloaded. C# has more restrictions. For many compound operators, C# automatically works out the meaning of the operator from the meanings of the constituent

operators, whereas C++ allows direct overload. For example, in C++, you can overload + and separately overload +=. In C# you can only overload +. The compiler will always use your overload of + to work out automatically the meaning of += for that class or struct.

The following operators can be overloaded in C# as well as C++:

- The binary arithmetic operators + - * /%
- The unary operators ++ and -- (prefix version only)
- The comparison operators !=, == <, <=, >, >=
- The bitwise operators &, |, ~, ^, !
- The Boolean values `true` and `false`

The following operators, which are used for overloading in C++, cannot be overloaded in C#:

- The arithmetic assignment operators *=, /=, +=, -=, %=. They are worked out by the compiler from the corresponding arithmetic operator and the assignment operator, which cannot be overloaded.
- The postfix increment operators. These are worked out by the compiler from the overloads of the corresponding prefix operators. They are implemented by calling the corresponding prefix operator overload, but returning the original value of the operand instead of the new value.
- The bitwise assignment operators &=, |=, ^=, >>=, and <<=.
- The Boolean operators &&, ||. These are worked out by the compiler from the corresponding bitwise operators.
- The assignment operator =. The meaning of this operator in C# is fixed.

There is also a restriction that the comparison operators must be overloaded in pairs — in other words, if you overload == you must overload != and vice versa. Similarly, if you overload one of < or >, you must overload both operators; likewise for <= and >=. The reason for this is to ensure consistent support for any database types that might have the value `null`, and for which, therefore, for example, == does not necessarily have the opposite effect to !=.

After you have established that the operator you want to overload is one that you can overload in C#, the syntax for actually defining the overload is much easier than the corresponding syntax in C++. The only points you need to be careful of in overloading C# operators are that they must always be declared as static members of a class. This contrasts with C++, which gives you the options to define your operator as a static member of the class, an instance member of the class but taking one parameter fewer, or as a function that is not a member of a class at all.

The reason that defining operator overloads is so much simpler in C# actually has nothing to do with the operator overloads themselves. It is because of the way that C# memory management naturally works to help you out. Defining operator overloads in C++ is an area that is filled with traps to catch the unwary. Consider, for example, an attempt to overload the addition operator for a class in C++. (Assume for this that `CMyClass` has a member, `x`, and adding instances means adding the `x` members.) The code might look something like this (assuming the overload is inline):

```
static CMyClass operator + (const CMyClass &lhs, const CMyClass &rhs)
{
   CMyClass Result;
   Result.x = lhs.x + rhs.x;
   return Result;
}
```

Note that the parameters are both declared as `const` and passed by reference, in order to ensure optimum efficiency. This by itself isn't too bad. However, in this case you need to create a temporary `CMyClass` instance inside the operator overload in order to return a result. The final `return Result` statement looks innocuous, but it will only compile if an assignment operator is available to copy `Result` out of the function, and works in an appropriate way. If you've defined your own copy constructor for `CMyClass`, you might also need to define the assignment operator yourself to make sure assignment behaves appropriately. That in itself is not a trivial task — if you don't use references correctly when defining it, it's very easy to define by accident one that recursively calls itself until you get a stack overflow! Put bluntly, overloading operators in C++ is not a task for inexperienced programmers! It's easy to see why Microsoft decided not to allow certain operators to be overloaded in C#.

In C# the picture is very different. There's no need to explicitly pass by reference, because C# classes are reference variables (and for structs, passing by reference tends to degrade rather than help performance), and returning a value is a breeze. Whether it's a class or a struct, you simply return the value of the temporary result, and the C# compiler ensures that either the member fields in the result are copied (for value types) or the address is copied (for reference types). The only disadvantage is that you cannot use the `const` keyword to get the extra compiler check that makes sure that the operator overload doesn't modify the parameters for a class. Also, C# doesn't give you the inline performance enhancements of C++:

```
Public static MyClass operator + (MyClass lhs, MyClass rhs)
{
   MyClass Result = new MyClass();
   Result.x = lhs.x + rhs.x;
   return Result;
}
```

Indexers

C# doesn't strictly permit `[]` to be overloaded. However, it permits you to define something called an *indexer* for a class, which gives the same effect.

The syntax for defining an indexer is very similar to that for a property. Suppose that you want to be able to treat instances of `MyClass` as an array, where each element is indexed with an `int` and returns a `long`. Then you would write:

```
class MyClass
{
   public long this[int x]
   {
      get
      {
         // code to get element
      }
      set
```

```
        {
            // code to set element. eg. X = value;
        }
    }
    // etc.
```

The code inside the `get` block is executed whenever the expression `Mine[x]` appears on the right-hand side of an expression (assuming `Mine` is an instance of `MyClass` and `x` is an `int`), whereas the `set` block is executed whenever `Mine[x]` appears on the left side of an expression. The `set` block cannot return anything, and uses the `value` keyword to indicate the quantity that appears on the right-hand side of the expression. The `get` block must return the same data type as that of the indexer.

It is possible to overload indexers to take any data type in the square brackets, or any number of arguments, allowing the effect of multidimensional arrays.

User-defined casts

Just as for indexers and `[]`, C# does not formally regard `()` as an operator that can be overloaded. However, it does permit the definition of user-defined casts, which have the same effect. For example, suppose you have two classes (or structs) called `MySource` and `MyDest`, and you want to define a cast from `MySource` to `MyDest`. The syntax looks like this:

```
public static implicit operator MyDest (MySource Source)
{
    // code to do cast. Must return a MyDest instance
}
```

The cast must be defined as a static member of either the `MyDest` or the `MySource` class. It must also be declared as either `implicit` or `explicit`. If you declare it as implicit, the cast can be used implicitly, like this:

```
MySource Source = new MySource();
MyDest Dest = Source;
```

If you declare it as explicit, the cast can only be used explicitly:

```
MySource Source = new MySource();
MyDest Dest = (MyDest) Source;
```

You should define implicit casts for conversions that will always work, and explicit casts for conversions that might fail by losing data or causing an exception to be thrown.

Just as in C++, if the C# compiler is faced with a request to convert between data types for which no direct cast exists, it will seek to find the best route using the casts it has available. The same issues as in C++ apply concerning the ensurance that your casts are intuitive and that different routes to achieve any conversion don't give incompatible results.

C# does not permit you to define casts between classes that are derived from each other. Such casts are already available — implicitly from a derived class to a base class and explicitly from a base class to a derived class.

Note that if you attempt to cast from a base class reference to a derived class reference, and the object in question is not an instance of the derived class (or anything derived from it), an exception will be thrown. In C++, it is not difficult to cast a pointer to an object to the "wrong" class of object. That is simply not possible in C# using references. For this reason, casting in C# is considered safer than in C++:

```
// assume MyDerivedClass is derived from MyBaseClass
MyBaseClass MyBase = new MyBaseClass();
MyDerivedClass MyDerived = (MyDerivedClass) MyBase; // this will result
// in an exception being thrown
```

If you don't want to try to cast something to a derived class, but don't want an exception to be thrown, you can use the `as` keyword. Using `as` if the cast fails simply returns `null`:

```
// assume MyDerivedClass is derived from MyBaseClass
MyBaseClass MyBase = new MyBaseClass();
MyDerivedClass MyDerived as (MyDerivedClass) MyBase; // this will
// return null
```

Arrays

Arrays are one area in which a superficial syntax similarity between C++ and C# hides the fact that what is actually going on behind the scenes is very different in the two languages. In C++, an array is essentially a set of variables packed together in memory and accessed via a pointer. In C#, on the other hand, an array is an instance of the base class `System.Array`, and is therefore a full-blown object stored on the heap under the control of the garbage collector. C# uses a C++-type syntax to access methods on this class in a way that gives the illusion of accessing arrays. The downside to this approach is that the overhead for arrays is greater than that for C++ arrays, but the advantage is that C# arrays are more flexible and easier to code around. As an example, C# arrays all have a property, `Length`, that gives the number of elements in the array, saving you from having to store this separately. C# arrays are also much safer to use — for example, the index bounds checking is performed automatically.

If you do want a simple array with none of the overhead of the `System.Array` class, it is still possible to do this in C#, but you'll need to use pointers and unsafe code blocks.

One-dimensional arrays

For one-dimensional arrays (C# terminology: arrays of *rank* 1), the syntax to access an array in the two languages is identical, with square brackets used to indicate elements of arrays. Arrays are also zero-indexed in both languages.

For example, to multiply each element in an array of `floats` by 2, you have to use this code:

```
// array declared as an array of floats
// this code works in C++ and C# without any changes
for (int i=0; i<10; i++)
array[i] *= 2.0f;
```

As mentioned earlier, however, C# arrays support the property `Length`, which can be used to find out how many elements are in it:

```
// array declared as an array of floats
// this code compiles in C# only
for (int i=0; i<array.Length; i++)
   array[i] *= 2.0f;
```

In C# you can also use the `foreach` statement to access elements of an array, as discussed earlier.

The syntax for declaring arrays is slightly different in C#, however, because C# arrays are always declared as reference objects:

```
double [] array; // Simply declares a reference without actually
// instantiating an array.
array = new double[10]; // Actually instantiates a System.Array object,
// and gives it size 10.
```

Or combining these statements, you might write:

```
double [] array = new double[10];
```

Notice that the array is only sized with its instance. The declaration of the reference simply uses the square brackets to indicate that the dimension (rank) of the array is one. In C#, rank is considered part of the type of the array, whereas the number of elements is not.

The nearest C++ equivalent to the preceding definition would be

```
double *pArray = new double[10];
```

This C++ statement actually gives a fairly close analogy, because both C++ and C# versions are allocated on the heap. Note that the C++ version is just an area of memory that contains ten `doubles`, whereas the C# version instantiates a full-blown object. The simpler stack version of C++

```
double pArray[10];
```

doesn't have a C# counterpart that uses actual C# arrays, although the C# `stackalloc` statement can achieve the equivalent to this statement using pointers. This is discussed later in the section on unsafe code.

Arrays in C# can be explicitly initialized when instantiated:

```
double [] array = new double[10]
{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
```

A shortened form exists too:

```
double [] array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
```

If an array is not explicitly initialized, the default constructor will automatically be called on each of its elements. (Elements of arrays are formally regarded as member fields of a class.) This behavior is very different from C++, which does not allow any kind of automatic initialization of arrays allocated with `new` on the heap (though C++ does allow this for stack-based arrays).

Multidimensional arrays

C# departs significantly from C++ in multidimensional arrays, because C# supports both rectangular and jagged arrays.

A rectangular array is a true grid of numbers. In C#, this is indicated by a syntax in which commas separate the number of elements in each dimension. Hence, for example, a two-dimensional rectangular array might be defined like this:

```
int [,] myArray2d;
myArray2d = new int[2,3] { {1, 0}, {3, 6}, {9, 12} };
```

The syntax here is a fairly intuitive extension of the syntax for one-dimensional arrays. The initialization list in the preceding code could be absent. For example:

```
int [,,] myArray3d = new int[2,3,2];
```

This causes the default constructor to be called on each element, initializing each `int` to zero. This particular example illustrates the creation of a three-dimensional array. The total number of elements in this array is 2×3×2 = 12. A characteristic of rectangular arrays is that each row has the same number of elements.

Elements of rectangular arrays are accessed using a similar syntax:

```
int x = myArray3d[1,2,0] + myArray2d[0,1];
```

C# rectangular arrays have no direct counterpart in C++. C# jagged arrays, however, correspond rather directly to multidimensional C++ arrays. For example, if you declare an array like this in C++:

```
int myCppArray[3][5];
```

what you are actually declaring is not really a 3×5 array, but an array of arrays — an array of size 3, each element of which is an array of size 5. This is perhaps clearer if you try to do the same thing dynamically — you would need to write:

```
int pMyCppArray = new int[3];
for (int i=0; i<3; i++)
   pMyCppArray[i] = new int[5];
```

It should be clear from this code that now there is no reason for each row to contain the same number of elements (though it happens to do so in this example). As an example of a jagged array in C++, which really does have different numbers of elements in each row, you might write:

```
int pMyCppArray = new int[3];
for (int i=0 ; i<3 ; i++)
   pMyCppArray[i] = new int[2*i + 2];
```

The respective rows of this array have dimensions 2, 4, and 6.

C# achieves the same result in the same manner, though in the C# case, the syntax indicates the numbers of dimensions more explicitly:

```
int [][] myJaggedArray = new int[3][];
for (int i=0; i<3; i++)
   myJaggedArray[i] = new int[2*i + 2];
```

Accessing members of a jagged array follows exactly the same syntax as for C++:

```
int x = myJaggedArray[1][3];
```

This code shows a jagged array with rank 2. Just as in C++, however, you can define a jagged array with whatever rank you want — you just need to add more square brackets to its definition.

Bounds checking

One area in which the object-oriented nature of C# arrays becomes apparent is bounds checking. If you ever attempt to access an array element in C# by specifying an index that is not within the array bounds, this will be detected at runtime, and an `IndexOutOfBoundsException` error will be thrown. In C++, this does not happen, and subtle runtime bugs can result. Once again, C# goes for extra checking against bugs at the expense of performance. Although you might expect this to result in a loss of performance, it does have the benefit that the .NET runtime is able to check through code to ensure that it is safe, in the sense that it will not attempt to access any memory beyond that allocated for its variables. This allows performance benefits, because different applications can, for example, be run in the same process and you can still be certain that those applications will be isolated from each other. There are also security benefits, because it is possible to predict more accurately what a given program will or won't attempt to do.

On the other hand, it's not uncommon for C++ programmers to use any of the various array wrapper classes in the standard library or in MFC in preference to raw arrays, in order to gain the same bounds checking and various other features — although in this case without the performance and security benefits associated with being able to analyze the program before it is executed.

Resizing arrays

C# arrays are dynamic insofar as you can specify the number of elements in each dimension at compile time (just as with dynamically allocated arrays in C++). However, it is not possible to resize them after they have been instantiated. If you need that kind of functionality, you'll have to look at some of the other related classes in the `System.Collections` namespace in the base class library, such as `System.Collections.ArrayList`. However, in this regard C# is no different from C++. Raw C++ arrays do not allow resizing, but a number of standard library classes are available to provide that feature.

Enumerations

In C#, it is possible to define an enumeration using the same syntax as in C++:

```
// valid in C++ or C#
enum TypeOfBuilding {Shop, House, OfficeBlock, School};
```

Note, however, that the trailing semicolon in C# is optional, because an enumeration definition in C# is effectively a struct definition, and struct definitions do not need trailing semicolons:

```
// valid in C# only
enum TypeOfBuilding {Shop, House, OfficeBlock, School}
```

However, in C# the enumeration must be named, whereas in C++, providing a name for the enumeration is optional. Just as in C++, C# numbers the elements of the array upwards from zero, unless you specify that an element should have a particular value:

```
enum TypeOfBuilding {Shop, House=5, OfficeBlock, School=10}
// Shop will have value 0, OfficeBlock will have value 6
```

The way that of accessing the values of the elements is different in C#, because in C# you must specify the name of the enumeration:

```
C++ syntax:

TypeOfBuilding MyHouse = House;

C# syntax:

TypeOfBuilding MyHouse = TypeOfBuilding.House;s
```

You might regard this as a disadvantage because the syntax is more cumbersome, but this actually reflects the fact that enumerations are far more powerful in C#. In C#, each enumeration is a fully fledged struct in its own right (derived from `System.Enum`), and therefore has certain methods available. In particular, for any enumerated value it is possible to do this:

```
TypeOfBuilding MyHouse = TypeOfBuilding.House;
string Result = MyHouse.ToString(); // Result will contain "House"
```

This is something almost impossible to achieve in C++. You can also go the other way in C#, using the static `Parse()` method of the `System.Enum` class, although the syntax is a little more awkward:

```
TypeOfBuilding MyHouse = (TypeOfBuilding)Enum.Parse(typeof(TypeOfBuilding),
"House", true);
```

`Enum.Parse()` returns an object reference, and so must be explicitly cast (unboxed) back to the appropriate `enum` type. The first parameter to `Parse()` is a `System.Type` object that describes which enumeration the string should represent. The second parameter is the string, and the third parameter indicates whether the case should be ignored. A second overload omits the third parameter and does not ignore the case.

C# also allows you to select the underlying data type used to store an enumerator:

```
enum TypeOfBuilding : short {Shop, House, OfficeBlock, School};
```

If you do not specify a type, the compiler will assume a default of `int`.

Exceptions

Exceptions are used in the same way in C# as in C++, apart from the following two differences:

- C# defines the `finally` block, which contains code that is always executed at the end of the `try` block, regardless of whether any exception was thrown. The lack of this feature in C++ has been a common cause of complaint among C++ developers. The `finally` block is executed as soon

as control leaves a `catch` or `try` block, and typically contains clean-up code for resources allocated in the `try` block.

- In C++ the class thrown in the exception may be any class. C#, however, requires that the exception be a class derived from `System.Exception`.

The rules for program flow through `try` and `catch` blocks are identical in C++ and C#. The syntax used is also identical, except for one difference — in C# a `catch` block that does not specify a variable to receive the exception object is denoted by the `catch` statement on its own:

```
C++ syntax:

catch (...)
{

C# syntax:

catch
{
```

In C#, this kind of `catch` statement can be useful to catch exceptions that are thrown by code written in other languages (and which therefore might not be derived from `System.Exception`. The C# compiler will flag an error if you attempt to define an exception object that isn't, but that isn't the case for other languages!).

The full syntax for `try...catch...finally` in C# looks like this:

```
try
{
   // normal code
}
catch (MyException e) // MyException derived from System.Exception
{
   // error handling code
}
   // optionally further catch blocks
finally
{
   // clean up code
}
```

Note that the `finally` block is optional. It is also permitted to have no `catch` blocks — in which case the `try...finally` construct simply serves as a way of ensuring that the code in the `finally` block is always executed when the `try` block exits. This might be useful if the `try` block contains several `return` statements and if you want some cleaning up of resources to be done before the method actually returns.

Pointers and Unsafe Code

Pointers can be declared in C# and they are very similar to C++. However, they can be declared and used only in an *unsafe* code block.

You can declare any method as unsafe:

```
public unsafe void MyMethod()
{
```

Alternatively, you can declare any class or struct as unsafe:

```
unsafe class MyClass
{
```

Declaring a class or struct as unsafe means that all members are regarded as unsafe. You can also declare any member field (but not local variables) as unsafe, if you have a member field of a pointer type:

```
private unsafe int* pX;
```

It is also possible to mark a block statement as unsafe:

```
unsafe
{
   // statements that use pointers
}
```

The syntax for declaring, accessing, dereferencing, and performing arithmetic operations on pointers is the same as in C++:

```
// This code would compile in C++ or C#, and has the same effect in both
// languages
int X = 10, Y = 20;
int *pX = &X;
*pX = 30;
pX = &Y;
++pX; // adds sizeof(int) to pX
```

Note the following points, however:

- In C# it is not permitted to dereference `void*` pointers, nor can you perform pointer arithmetic operations on `void*` pointers. The `void*` pointer syntax has been retained for backward compatibility, to call external API functions that are not .NET-aware and which require `void*` pointers as parameters.
- Pointers cannot point to reference types (classes or arrays). Nor can they point to structs that contain embedded reference types as members. This is really an attempt to protect data that is used by the garbage collector and by the .NET runtime (though in C#, just as in C++, once you start using pointers you can almost always find a way around any restriction by performing arithmetic operations on pointers and then dereferencing).
- Besides declaring the relevant parts of your code as unsafe, you also need to specify the `/unsafe` flag to the compiler when compiling code that contains pointers.
- Pointers cannot point to any variables that are embedded in reference data types (for example, members of classes) unless declared inside a `fixed` statement.

Fixing data on the heap

It is permitted to assign the address of a value type to a pointer even if that value type is embedded as a member field in a reference type. However, such a pointer must be declared inside a `fixed` statement. The reason for this is that reference types can be moved around on the heap at any time by the garbage collector. The garbage collector is aware of C# references and can update them as necessary, but it is not aware of pointers. Hence, if a pointer points to a class member on the heap and the garbage collector moves the entire class instance, the pointer will end up pointing to the wrong address. The `fixed` statement prevents the garbage collector from moving the specified class instance for the duration of the `fixed` block, ensuring the integrity of pointer values:

```
class MyClass
{
   public int X;
   // etc.
}

// Elsewhere in your code ...
MyClass Mine = new MyClass();
// Do processing
fixed(int *pX = Mine.X)
{
   // Can use pX in this block.
}
```

It is possible to nest `fixed` blocks in order to declare more than one pointer. You can also declare more than one pointer in a single `fixed` statement, provided both pointers have the same reference type:

```
fixed(int *pX = Mine.X, *pX2 = Mine2.X)
{
```

Declaring arrays on the stack

C# provides an operator called `stackalloc`, which can be used in conjunction with pointers to declare a low-overhead array on the stack. The allocated array is not a full C#-style `System.Array` object, but a simple array of numbers analogous to a one-dimensional C++ array. The elements of this array are not initialized and are accessed using the same syntax as in C++ by applying square brackets to the pointer.

The `stackalloc` operator requires specification of the data type and the number of elements for which space is required:

```
C++ syntax:

unsigned long pMyArray[20];

C# syntax:

ulong *pMyArray = stackalloc ulong [20];
```

Note, however, that although these arrays are exactly analogous, the C# version allows the size to be determined at runtime:

```
int X;
// Initialize X
ulong *pMyArray = stackalloc ulong [X];
```

Interfaces

Interfaces are an aspect of C# with no direct equivalent in ANSI C++, although Microsoft has introduced interfaces in C++ with a Microsoft-specific keyword. The idea of an interface evolved from COM interfaces, which are intended as contracts, indicating which methods and properties an object implements.

An interface in C# is not quite the same as a COM interface, because it does not have an associated GUID, does not derive from `IUnknown` and does not have associated registry entries (although it is possible to map a C# interface onto a COM interface). A C# interface is simply a set of definitions for functions and properties. It can be considered as analogous to an abstract class and is defined using a similar syntax to a class:

```
interface IMyInterface
{
   void MyMethod(int X);
}
```

You'll notice, however, the following syntactical differences from a class definition:

- The methods do not have access modifiers.
- Methods can never be implemented in an interface.
- Methods cannot be declared as virtual or explicitly as an abstract. The choice of how to implement methods is the responsibility of any class that implements this interface.

A class implements an interface by deriving from it. Although a class can only be derived from one other class, it can also derive from as many interfaces as you want. If a class implements an interface, it must supply implementations of all methods defined by that interface:

```
class MyClass : MyBaseClass, IMyInterface, IAnotherInterface // etc
{
   public virtual void MyMethod(int X)
   {
      // implementation
   }
     // etc.
```

This example implements `MyMethod` as a virtual method with public access.

Interfaces can also derive from other interfaces, in which case the derived interface contains its own methods as well as those of the base interface:

```
interface IMyInterface : IBaseInterface
```

You can check that an object implements an interface either by using the `is` operator or by using the `as` operator to cast it to that interface. Alternatively, you can cast it directly, but in that case you'll get an exception if the object doesn't implement the interface, so that approach is only advisable if you know the cast will succeed. The reference to the interface can be used to call methods on that interface (the implementation being supplied by the class instance):

```
IMyInterface MyInterface;
MyClass Mine = new MyClass();
MyInterface = Mine as IMyInterface;
if (MyInterface != null)
   MyInterface.MyMethod(10);
```

The main uses of interfaces are as follows:

- For interoperability and backward compatibility with COM components.
- To serve as contracts for other .NET classes. An interface can be used to indicate that a class implements certain features. For example, the C# `foreach` loop works internally by checking that the class to which it is applied implements the `IEnumerable` interface, and then subsequently calling methods defined by that interface.

Delegates

A delegate in C# has no direct equivalent in C++ and performs the same task as a C++ function pointer. The idea of a delegate is that the method pointer is wrapped in a specialized class, along with a reference to the object against which the method is to be called (for an instance method, or the null reference for a static method). This means that, unlike a C++ function pointer, a C# delegate contains enough information to call an instance method.

Formally, a delegate is a class derived from the class `System.Delegate`. Hence instantiating a delegate involves two stages: defining this derived class, then declaring a variable of the appropriate type. The definition of a delegate class includes details of the full signature (including the return type) of the method that the delegate wraps.

The main use for delegates is for passing around references to methods and invoking them. References to methods cannot be passed around directly, but they can be passed around inside the delegate. The delegate ensures type safety by preventing a method with the wrong signature from being invoked. The method contained by the delegate can be invoked by syntactically invoking the delegate. The following code demonstrates the general principles.

First, you need to define the delegate class:

```
// Define a delegate class that represents a method that takes an int and
// returns void
delegate void MyOp(int X);
```

Next, for the purposes of this example, you declare a class that contains the method to be invoked:

```
// Later _ a class definition
class MyClass
{
   void MyMethod(int X)
   {
      // etc.
   }
}
```

Then, later on, perhaps in the implementation of some other class, you have the method that is to be passed a method reference via a delegate:

```
void MethodThatTakesDelegate(MyOp Op)
{
   // call the method passing in value 4
   Op(4);
}
  // etc.
```

And finally, the code that actually uses the delegate:

```
MyClass Mine = new MyClass();
// Instantiate a MyOp delegate. Set it to point to the MyMethod method
// of Mine.
MyOp DoIt = new MyOp(Mine.MyMethod);
```

Once this delegate variable is declared, you can invoke the method via the delegate:

```
DoIt();
```

Or pass it to another method:

```
MethodThatTakesDelegate(DoIt);
```

In the particular case that a delegate represents a method that returns a `void`, this delegate is a multicast delegate and can simultaneously represent more than one method. Invoking the delegate causes all the methods it represents to be invoked in turn. The + and += operators can be used to add a method to a delegate, and _ and -= can be used to remove a method that is already in the delegate. Delegates are explained in more detail in Chapter 6, "Delegates and Events."

Events

Events are specialized forms of delegates used to support the callback event notification model. An event is a delegate that has this signature:

```
delegate void EventClass(obj Sender, EventArgs e);
```

This is the signature any event handler that is called back must have. `Sender` is expected to be a reference to the object that raised the event, whereas `System.EventArgs` (or any class derived from

`EventArgs` — this is also permitted as a parameter) is the class used by the .NET runtime to pass generic information concerning the details of an event.

The special syntax for declaring an event is this:

```
public event EventClass OnEvent;
```

Clients use the `+=` syntax of multicast delegates to inform the event that they want to be notified:

```
// EventSource refers to the class instance that contains the event
EventSource.OnEvent += MyHandler;
```

The event source then simply invokes the event when required, using the same syntax as demonstrated previously for delegates. Because the event is a multicast delegate, all the event handlers will be called in the process.

```
OnEvent(this, new EventArgs());
```

Events are dealt with in detail in Chapter 6.

Attributes

Attributes are a concept that has no equivalent in ANSI C++, though they are supported by the Microsoft C++ compiler as a Windows-specific extension. In the C# version, they are .NET classes that are derived from `System.Attribute`. They can be applied to various elements in C# code (classes, enums, methods, parameters, and so on) to generate extra documentation information in the compiled assembly. In addition, certain attributes are recognized by the C# compiler and will have an effect on the compiled code.

The following list describes some of the available attributes:

- `DllImport` — Indicates that a method is defined in an external DLL.
- `StructLayout` — Permits the contents of a struct to be laid out in memory. Allows the same functionality as a C++ union.
- `Obsolete` — Generates a compiler error or warning if this method is used.
- `Conditional` — Forces a conditional compilation. This method and all references to it will be ignored unless a particular preprocessor symbol is present.

A large number of other attributes are available, and it is also possible to define your own custom ones. Use of attributes is discussed in Chapter 11, "Reflection."

The syntax of attributes is that they appear immediately before the object to which they apply, in square brackets. This is the same syntax as for Microsoft C++ attributes:

```
[Conditional("Debug")]
void DisplayValuesOfImportantVariables()
{
   // etc.
```

Templates and Generics

Although .NET 1.1 didn't have an equivalent of C++ templates, the new .NET 2.0 generics can be compared with this C++ feature. Similar to C++ it is possible to define types and methods where the type of members is undefined when the generic type is defined. This is a very useful construct with collections.

A generic type is created with C++ using the `template` keyword and angle brackets. Within the angle brackets a comma-separated list of template parameters is defined. The following example shows the types `T` and `U`. The type `T` is used as a member variable, and with the method `Method1()` as a parameter and return type, and a pointer to U is returned with `Method2()`:

```
template<typename T, typename U>
class MyTemplate
{
private:
   T element;
public:
   T Method1(T obj)
   {
      return obj;
   }
   U* Method2()
   {
      U* obj = new U();
      return obj;
   }
};
```

A template class is instantiated by defining the types that should be used for the generic types within the angle brackets. Here the classes `A` and `B` are used instead of `T` and `U`:

```
MyTemplate<A, B>* pObj = new MyTemplate<A, B>();
A a;
a.a1 = 33;
pObj->Method(a);
B* b1 = pObj->Method2();
delete b1;
delete pObj;
```

The syntax for using generics with C# requires angle brackets. This is very similar to C++ templates, only the `template` keyword is not used. The angle brackets are all that's needed to specify a generic type:

```
public class MyGeneric<T>
{
   private T element;

   public T Method(T parm)
   {
      return null;
   }
}
```

Besides the syntax there's another big difference between templates and generics. Whereas templates are a C++ language construct, and you are required to have the source code of a template for instantiating templates, generics are available with IL code and thus can be instantiated from other programming languages that support generics (for example, Visual Basic).

With C++ generic types the requirements for the generic type are defined by the methods and operators that are used within the template class implementation. The C++ compiler cannot verify code correctness when defining the template class. The compiler does not complain with the following template definition; it only complains when the template class is instantiated with a type that doesn't implement the method `DoesNotExist()` in case the method `Method()` is invoked:

```
template<typename T>
class MyTemplate
{
public:
   void Method()
   {
      T obj;
      obj.DoesNotExist();
   }
};
```

This behavior is very different with C#. With C# the requirements for generic types are defined with the `where` keyword. The `where` keyword specifies that the type must either derive from a specific base class or implement an interface. The `where` keyword also allows specifying that the type must implement a default constructor. The sample code shows the generic class `MyGeneric` where the type `T` must implement the interface `IDemo` because the `Demo()` method is invoked and requires a default constructor with the constructor constraint. Without specifying these constraints the C# compiler doesn't allow invoking methods and creating instances within a generic class.

```
public interface IDemo
{
   void Demo();
}

public class MyGeneric<T>
   where T : IDemo, new()
{
   public void Method()
   {
      T obj = new T();
      obj.Demo();
   }
}
```

Although these constraints allow for better compile-time errors, this technique has some limitations. With .NET 2.0 it is not possible to define other than default constructors, and it is not possible to define required operator overloads. Future versions may allow for extended syntax to add this functionality to generics.

Preprocessor Directives

C# supports preprocessor directives in the same way as C++, except that there are far fewer of them. In particular, C# does not support the commonly used `#include` of C++. (It's not needed because C# does not require forward declarations.)

The syntax for preprocessor directives is the same in C# as in C++. The following table lists the directives that are supported by C#.

Directive	Meaning
`#define/#undef`	Same as C++, except that they must appear at the start of the file, before C# code.
`#if/#elif/#else/#endif`	Same as C++ `#ifdef/#elif/#else/#endif`.
`#line`	Same as C++ `#line`.
`#warning/#error`	Same as C++ `#warning/#error`.
`#region/#endregion`	Marks off a block of code as a region. Regions are recognized by certain editors (such as the folding editors of Visual Studio) and so can be used to improve the layout of code presented to the user while editing.

Summary

This appendix looked at the differences between C++ and C# from the viewpoint of a developer already familiar with C++.

As you've seen with the C++ syntax, C# has been influenced a lot by C++. C# has many similarities to C++. Language syntax, program flow statements, and defining classes are very similar to C++. However, where bugs happened with C++ programming, C# got a new behavior (for example, the `switch` statement).

C# adds some features that cannot be found with C++. Properties, delegates, and events are some examples. Memory management is very different between C++ and C# because of the managed environment offered by the CLR.