**The Preprocessor**

* C# defines several preprocessor directives, which affect the way that your program’s source file is interpreted by the compiler.
* These directives affect the text of the source file in which they occur, prior to the translation of the program into object code.
* The term preprocessor directive comes from the fact that these instructions were traditionally handled by a separate compilation phase called the preprocessor.
* Today’s modern compiler technology no longer requires a separate preprocessing stage to handle the directives, but the name has stuck.
* C# defines the following preprocessor directives:

|  |  |  |  |
| --- | --- | --- | --- |
| #define | #elif | #else | #endif |
| #endregion | #error | #if | #line |
| #pragma | #region | #undef | #warning |

* All preprocessor directives begin with a # sign.
* In addition, each preprocessor directive must be on its own line.
* Given C#’s modern, object-oriented architecture, there is not as much need for the preprocessor directives as there is in older languages. Nevertheless, they can be of value from time to time, especially for conditional compilation. Each directive is examined in turn.

**#define**

* The #define directive defines a character sequence called a symbol.
* The existence or nonexistence of a symbol can be determined by #if or #elif and is used to control compilation.
* Here is the general form for #define:

**#define symbol**

* Notice that there is no semicolon in this statement.
* There may be any number of spaces between the #define and the symbol, but once the symbol begins, it is terminated only by a newline.
* For example, to define the symbol EXPERIMENTAL, use this directive:

**#define EXPERIMENTAL**

* NOTE: In C/C++ you can use #define to perform textual substitutions, such as defining a name for a value, and to create function-like macros.
* C# does not support these uses of #define.
* In C#, #define is used only to define a symbol.

**#if and #endif**

* The #if and #endif directives enable conditional compilation of a sequence of code based upon whether an expression involving one or more symbols evaluates to true.
* A symbol is true if it has been defined. It is false otherwise.
* Thus, if a symbol has been defined by a #define directive, it will evaluate as true.
* The general form of #if is

**#if symbol-expression**

**statement sequence**

**#endif**

* If the expression following #if is true, the code that is between it and #endif is compiled.
* Otherwise, the intervening code is skipped.
* The #endif directive marks the end of an #if block.
* A symbol expression can be as simple as just the name of a symbol.
* You can also use these operators in a symbol expression: !, = =, !=, &&, and ||. Parentheses are also allowed.

Here’s an example:

// Demonstrate #if, #endif, and #define.

**#define EXPERIMENTAL**

**using System;**

**class Test**

{

**public static void Main(String[] args)**

{

#if EXPERIMENTAL

Console.WriteLine("Compiled for experimental version.");

#endif

Console.WriteLine("This is in all versions.");

}

}

This program displays the following:

* The program defines the symbol EXPERIMENTAL. Thus, when the #if is encountered, the symbol expression evaluates to true, and the first WriteLine( ) statement is compiled.
* If you remove the definition of EXPERIMENTAL and recompile the program, the first WriteLine( ) statement will not be compiled, because the #if will evaluate to false.
* In all cases, the second WriteLine( ) statement is compiled because it is not part of the #if block.
* As explained, you can use a symbol expression in an #if.

**For example,**

// Use a symbol expression.

**#define EXPERIMENTAL**

**#define TRIAL**

**using System;**

**class Test**

{

**public static void Main(String[] args)**

{

#if EXPERIMENTAL

Console.WriteLine("Compiled for experimental version.");

#endif

#if EXPERIMENTAL && TRIAL

Console.Error.WriteLine("Testing experimental trial version.");

#endif

Console.WriteLine("This is in all versions.");

}

}

* In this example, two symbols are defined, EXPERIMENTAL and TRIAL.
* The second WriteLine( ) statement is compiled only if both are defined.
* You can use the ! to compile code when a symbol is not defined.

**#if !EXPERIMENTAL**

**Console.WriteLine("Code is not experimental!");**

**#endif**

The call to WriteLine( ) will be compiled only if EXPERIMENTAL has not been defined.

**#else and #elif**

The #else directive works much like the else that is part of the C# language: It establishes an alternative if #if fails.

**The previous example can be expanded as shown here:**

// Demonstrate #else.

**#define EXPERIMENTAL**

**using System;**

**class Test**

{

**static void Main()**

{

#if EXPERIMENTAL

Console.WriteLine("Compiled for experimental version.");

#else

Console.WriteLine("Compiled for release.");

#endif

#if EXPERIMENTAL && TRIAL

Console.Error.WriteLine("Testing experimental trial version.");

#else

Console.Error.WriteLine("Not experimental trial version.");

#endif

Console.WriteLine("This is in all versions.");

}

}

* Since TRIAL is not defined, the #else portion of the second conditional code sequence is used.
* Notice that #else marks both the end of the #if block and the beginning of the #else block.
* This is necessary because there can only be one #endif associated with any #if.
* Furthermore, there can be only one #else associated with any #if.
* The #elif directive means “else if” and establishes an if-else-if chain for multiple compilation options.
* #elif is followed by a symbol expression.
* If the expression is true, that block of code is compiled and no other #elif expressions are tested. Otherwise, the next block in the series is checked.If no #elif succeeds, then if there is a #else, the code sequence associated with the #else is compiled. Otherwise, no code in the entire #if is compiled.

**The general form for #elif is**

#if symbol-expression

statement sequence

#elif symbol-expression

statement sequence

#elif symbol-expression

statement sequence

// . . .

#endif

**Here’s an example:**

// Demonstrate #elif.

**#define RELEASE**

**using System;**

**class Test**

{

public static void Main(String[] args)

{

#if EXPERIMENTAL

Console.WriteLine("Compiled for experimental version.");

#elif RELEASE

Console.WriteLine("Compiled for release.");

#else

Console.WriteLine("Compiled for internal testing.");

#endif

#if TRIAL && !RELEASE

Console.WriteLine("Trial version.");

#endif

Console.WriteLine("This is in all versions.");

}

}

**#undef**

* The #undef directive removes a previously defined symbol.
* That is, it “undefines” a symbol.
* The general form for #undef is

**#undef symbol**

Here’s an example:

**#define SMALL**

**#if SMALL**

// ...

**#undef SMALL**

// at this point SMALL is undefined.

* After the #undef directive, SMALL is no longer defined.
* #undef is used principally to allow symbols to be localized to only those sections of code that need them.

**#error**

* The #error directive forces the compiler to stop compilation. It is used for debugging.
* The general form of the #error directive is

**#error error-message**

* When the #error directive is encountered, the error message is displayed.
* For example, when the compiler encounters this line:

**#error This is a test error!**

* compilation stops and the error message “This is a test error!” is displayed.

**#warning**

* The #warning directive is similar to #error, except that a warning rather than an error is produced. Thus, compilation is not stopped.
* The general form of the #warning directive is

**#warning warning-message**

**#line**

* The #line directive sets the line number and filename for the file that contains the #line directive.
* The number and the name are used when errors or warnings are output during compilation.
* The general form for #line is

#line number “fi lename”

* where number is any positive integer, which becomes the new line number, and the optional filename is any valid file identifier, which becomes the new filename.
* #line is primarily used for debugging and special applications.
* #line allows two options.
* The first is default, which returns the line numbering to its original condition. It is used like this:

#line default

* The second is hidden. When stepping through a program, the hidden option allows a debugger to bypass lines between a #line hidden directive and the next #line directive that does not include the hidden option.

**#region and #endregion**

* The #region and #endregion directives let you define a region that will be expanded or collapsed when using outlining in the Visual Studio IDE.
* The general form is shown here:

#region text

// code sequence

#endregion text

* Here, text is an optional string.

**#pragma**

* The #pragma directive gives instructions, such as specifying an option, to the compiler.
* It has this general form:

**#pragma option**

* Here, option is the instruction passed to the compiler.
* In C# 3.0, there are two options supported by #pragma.
* The first is warning, which is used to enable or disable specific compiler warnings.
* It has these two forms:
  + **#pragma warning disable warnings**
  + **#pragma warning restore warnings**
* Here, warnings is a comma-separated list of warning numbers.
* To disable a warning, use the disable option.
* To enable a warning, use the restore option.
* For example, this #pragma statement disables warning 168, which indicates when a variable is declared but not used:

#pragma warning disable 168

* The second #pragma option is checksum. It is used to generate checksums for ASP.NET projects. It has this general form.

**#pragma checksum “fi lename” “{GUID}” “check-sum”**

* Here, filename is the name of the file, GUID is the globally unique identifier associated with filename, and check-sum is a hexadecimal number that contains the checksum.
* This string must contain an even number of digits.