Contents

C# preprocessor directives

1/23/2019 • 2 minutes to read • Edit Online

This section contains information about the following C# preprocessor directives:

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

See the individual topics for more information and examples.

Although the compiler doesn't have a separate preprocessor, the directives described in this section are processed as if there were one. They are used to help in conditional compilation. Unlike C and C++ directives, you cannot use these directives to create macros.

A preprocessor directive must be the only instruction on a line.

See also

- C# Reference
- C# Programming Guide

#if (C# Reference)

2/6/2019 • 2 minutes to read • Edit Online

When the C# compiler encounters an `#if` directive, followed eventually by an #endif directive, it compiles the code between the directives only if the specified symbol is defined. Unlike C and C++, you cannot assign a numeric value to a symbol. The #if statement in C# is Boolean and only tests whether the symbol has been defined or not. For example:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

You can use the operators == (equality) and != (inequality) only to test for true or false. True means the symbol is defined. The statement `#if DEBUG` has the same meaning as `#if (DEBUG == true)`. You can use the operators && (and), || (or), and ! (not) to evaluate whether multiple symbols have been defined. You can also group symbols and operators with parentheses.

Remarks

`#if`, along with the #else, #elif, #endif, #define, and #undef directives, lets you include or exclude code based on the existence of one or more symbols. This can be useful when compiling code for a debug build or when compiling for a specific configuration.

A conditional directive beginning with a `#if` directive must explicitly be terminated with a `#endif` directive.

`#define` lets you define a symbol. By then using the symbol as the expression passed to the `#if` directive, the expression evaluates to `true`.

You can also define a symbol with the -define compiler option. You can undefine a symbol with #undef.

A symbol that you define with `-define` or with `#define` doesn't conflict with a variable of the same name. That is, a variable name should not be passed to a preprocessor directive, and a symbol can only be evaluated by a preprocessor directive.

The scope of a symbol created with `#define` is the file in which it was defined.

The build system is also aware of predefined preprocessor symbols representing different target frameworks. They're useful when creating applications that can target more than one .NET implementation or version.

TARGET FRAMEWORKS	SYMBOLS
.NET Framework	`NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`
.NET Standard	`NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`
.NET Core	`NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`

Other predefined symbols include the DEBUG and TRACE constants. You can override the values set for the

project using `#define`. The DEBUG symbol, for example, is automatically set depending on your build configuration properties ("Debug" or "Release" mode).

Examples

The following example shows you how to define a MYTEST symbol on a file and then test the values of the MYTEST and DEBUG symbols. The output of this example depends on whether you built the project on Debug or Release configuration mode.

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

The following example shows you how to test for different target frameworks so you can use newer APIs when possible:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives
- How to: Compile Conditionally with Trace and Debug

#else (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#else` lets you create a compound conditional directive, so that, if none of the expressions in the preceding #if or (optional) #elif directives evaluate to `true`, the compiler will evaluate all code between `#else` and the subsequent `#endif`.

Remarks

#endif must be the next preprocessor directive after `#else`. See #if for an example of how to use `#else`.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#elif (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#elif` lets you create a compound conditional directive. The `#elif` expression will be evaluated if neither the preceding #if nor any preceding, optional, `#elif` directive expressions evaluate to `true`. If a `#elif` expression evaluates to `true`, the compiler evaluates all the code between the `#elif` and the next conditional directive. For example:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

You can use the operators `==` (equality), `!=` (inequality), `&&` (and), and `||` (or), to evaluate multiple symbols. You can also group symbols and operators with parentheses.

Remarks

`#elif` is equivalent to using:

```
#else
#if
```

Using `#elif` is simpler, because each `#if` requires a #endif, whereas a `#elif` can be used without a matching `#endif`.

See #if for an example of how to use `#elif`.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#endif (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#endif` specifies the end of a conditional directive, which began with the #if directive. For example,

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Remarks

A conditional directive, beginning with a `#if` directive, must explicitly be terminated with a `#endif` directive. See #if for an example of how to use `#endif`.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#define (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

You use `#define` to define a symbol. When you use the symbol as the expression that's passed to the #if directive, the expression will evaluate to `true`, as the following example shows:

```
#define DEBUG
```

Remarks

> **NOTE**
> The `#define` directive cannot be used to declare constant values as is typically done in C and C++. Constants in C# are best defined as static members of a class or struct. If you have several such constants, consider creating a separate "Constants" class to hold them.

Symbols can be used to specify conditions for compilation. You can test for the symbol with either #if or #elif. You can also use the ConditionalAttribute to perform conditional compilation.

You can define a symbol, but you cannot assign a value to a symbol. The `#define` directive must appear in the file before you use any instructions that aren't also preprocessor directives.

You can also define a symbol with the -define compiler option. You can undefine a symbol with #undef.

A symbol that you define with `-define` or with `#define` does not conflict with a variable of the same name. That is, a variable name should not be passed to a preprocessor directive and a symbol can only be evaluated by a preprocessor directive.

The scope of a symbol that was created by using `#define` is the file in which the symbol was defined.

As the following example shows, you must put `#define` directives at the top of the file.

```
#define DEBUG
//#define TRACE
#undef TRACE

using System;

public class TestDefine
{
    static void Main()
    {
#if (DEBUG)
        Console.WriteLine("Debugging is enabled.");
#endif

#if (TRACE)
     Console.WriteLine("Tracing is enabled.");
#endif
    }
}
// Output:
// Debugging is enabled.
```

For an example of how to undefine a symbol, see #undef.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives
- const
- How to: Compile Conditionally with Trace and Debug
- #undef
- #if

#undef (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#undef` lets you undefine a symbol, such that, by using the symbol as the expression in a #if directive, the expression will evaluate to `false`.

A symbol can be defined either with the #define directive or the -define compiler option. The `#undef` directive must appear in the file before you use any statements that are not also directives.

Example

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
#if DEBUG
        Console.WriteLine("DEBUG is defined");
#else
        Console.WriteLine("DEBUG is not defined");
#endif
    }
}
```

DEBUG is not defined

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#warning (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#warning` lets you generate a CS1030 level one compiler warning from a specific location in your code. For example:

```
#warning Deprecated code in this method.
```

Remarks

A common use of `#warning` is in a conditional directive. It is also possible to generate a user-defined error with #error.

Example

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#warning DEBUG is defined
#endif
    }
}
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#error (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#error` lets you generate a CS1029 user-defined error from a specific location in your code. For example:

```
#error Deprecated code in this method.
```

Remarks

A common use of `#error` is in a conditional directive.

It is also possible to generate a user-defined warning with #warning.

Example

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#error DEBUG is defined
#endif
    }
}
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#line (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#line` lets you modify the compiler's line numbering and (optionally) the file name output for errors and warnings.

The following example shows how to report two warnings associated with line numbers. The `#line 200` directive forces the next line's number to be 200 (although the default is #6) and until the next #line directive, the filename will be reported as "Special". The #line default directive returns the line numbering to its default numbering, which counts the lines that were renumbered by the previous directive.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

Compilation produces the following output:

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

Remarks

The `#line` directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with `#line`.

The `#line hidden` directive hides the successive lines from the debugger, such that when the developer steps through the code, any lines between a `#line hidden` and the next `#line` directive (assuming that it is not another `#line hidden` directive) will be stepped over. This option can also be used to allow ASP.NET to differentiate between user-defined and machine-generated code. Although ASP.NET is the primary consumer of this feature, it is likely that more source generators will make use of it.

A `#line hidden` directive does not affect file names or line numbers in error reporting. That is, if an error is encountered in a hidden block, the compiler will report the current file name and line number of the error.

The `#line filename` directive specifies the file name you want to appear in the compiler output. By default, the

actual name of the source code file is used. The file name must be in double quotation marks ("") and must be preceded by a line number.

A source code file can have any number of `#line` directives.

Example 1

The following example shows how the debugger ignores the hidden lines in the code. When you run the example, it will display three lines of text. However, when you set a break point, as shown in the example, and hit F10 to step through the code, you will notice that the debugger ignores the hidden line. Notice also that even if you set a break point at the hidden line, the debugger will still ignore it.

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#region (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#region` lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. In longer code files, it is convenient to be able to collapse or hide one or more regions so that you can focus on the part of the file that you are currently working on. The following example shows how to define a region:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Remarks

A `#region` block must be terminated with a #endregion directive.

A `#region` block cannot overlap with a #if block. However, a `#region` block can be nested in a `#if` block, and a `#if` block can be nested in a `#region` block.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#endregion (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#endregion` marks the end of a #region block. For example:

```
#region MyClass definition
class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives

#pragma (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#pragma` gives the compiler special instructions for the compilation of the file in which it appears. The instructions must be supported by the compiler. In other words, you cannot use `#pragma` to create custom preprocessing instructions. The Microsoft C# compiler supports the following two `#pragma` instructions:

#pragma warning

#pragma checksum

Syntax

```
#pragma pragma-name pragma-arguments
```

Parameters

`pragma-name`

The name of a recognized pragma.

`pragma-arguments`

Pragma-specific arguments.

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives
- #pragma warning
- #pragma checksum

#pragma warning (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

`#pragma warning` can enable or disable certain warnings.

Syntax

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Parameters

`warning-list`

A comma-separated list of warning numbers. The "CS" prefix is optional.

When no warning numbers are specified, `disable` disables all warnings and `restore` enables all warnings.

> **NOTE**
>
> To find warning numbers in Visual Studio, build your project and then look for the warning numbers in the **Output** window.

Example

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

See also

- C# Reference
- C# Programming Guide
- C# Preprocessor Directives
- C# Compiler Errors

#pragma checksum (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

Generates checksums for source files to aid with debugging ASP.NET pages.

Syntax

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Parameters

`"filename"`
The name of the file that requires monitoring for changes or updates.

`"{guid}"`
The Globally Unique Identifier (GUID) for the hash algorithm.

`"checksum_bytes"`
The string of hexadecimal digits representing the bytes of the checksum. Must be an even number of hexadecimal digits. An odd number of digits results in a compile-time warning, and the directive are ignored.

Remarks

The Visual Studio debugger uses a checksum to make sure that it always finds the right source. The compiler computes the checksum for a source file, and then emits the output to the program database (PDB) file. The debugger then uses the PDB to compare against the checksum that it computes for the source file.

This solution does not work for ASP.NET projects, because the computed checksum is for the generated source file, rather than the .aspx file. To address this problem, `#pragma checksum` provides checksum support for ASP.NET pages.

When you create an ASP.NET project in Visual C#, the generated source file contains a checksum for the .aspx file, from which the source is generated. The compiler then writes this information into the PDB file.

If the compiler encounters no `#pragma checksum` directive in the file, it computes the checksum and writes the value to the PDB file.

Example

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

See also

- C# Reference
- C# Programming Guide

- C# Preprocessor Directives