

Contents

Language Reference

[Configure language version](#)

[C# Keywords](#)

[C# Operators](#)

[C# Special Characters](#)

[C# Preprocessor Directives](#)

[C# Compiler Options](#)

[C# Compiler Errors](#)

[C# 6.0 draft Language Specification](#)

C# Reference

5/24/2018 • 2 minutes to read • [Edit Online](#)

This section provides reference material about C# keywords, operators, special characters, preprocessor directives, compiler options, and compiler errors and warnings.

In This Section

[C# Keywords](#)

Provides links to information about C# keywords and syntax.

[C# Operators](#)

Provides links to information about C# operators and syntax.

[C# Special Characters](#)

Provides links to information about special contextual characters in C# and their usage.

[C# Preprocessor Directives](#)

Provides links to information about compiler commands for embedding in C# source code.

[C# Compiler Options](#)

Includes information about compiler options and how to use them.

[C# Compiler Errors](#)

Includes code snippets that demonstrate the cause and correction of C# compiler errors and warnings.

[C# Language Specification](#)

Provides links to the latest versions of the C# language specification.

Related Sections

[C# Guide](#)

Provides a portal to Visual C# documentation.

[Using the Visual Studio Development Environment for C#](#)

Provides links to conceptual and task topics that describe the IDE and Editor.

[C# Programming Guide](#)

Includes information about how to use the C# programming language.

Select the C# language version

6/15/2018 • 3 minutes to read • [Edit Online](#)

The C# compiler defaults to the latest major version of the language that has been released. You may choose to compile any project using a new point release of the language. Choosing a newer version of the language enables your project to make use of the latest language features. In other scenarios, you may need to validate that a project compiles cleanly when using an older version of the language.

This capability decouples the decision to install new versions of the SDK and tools in your development environment from the decision to incorporate new language features in a project. You can install the latest SDK and tools on your build machine. Each project can be configured to use a specific version of the language for its build.

There are several ways to set the language version:

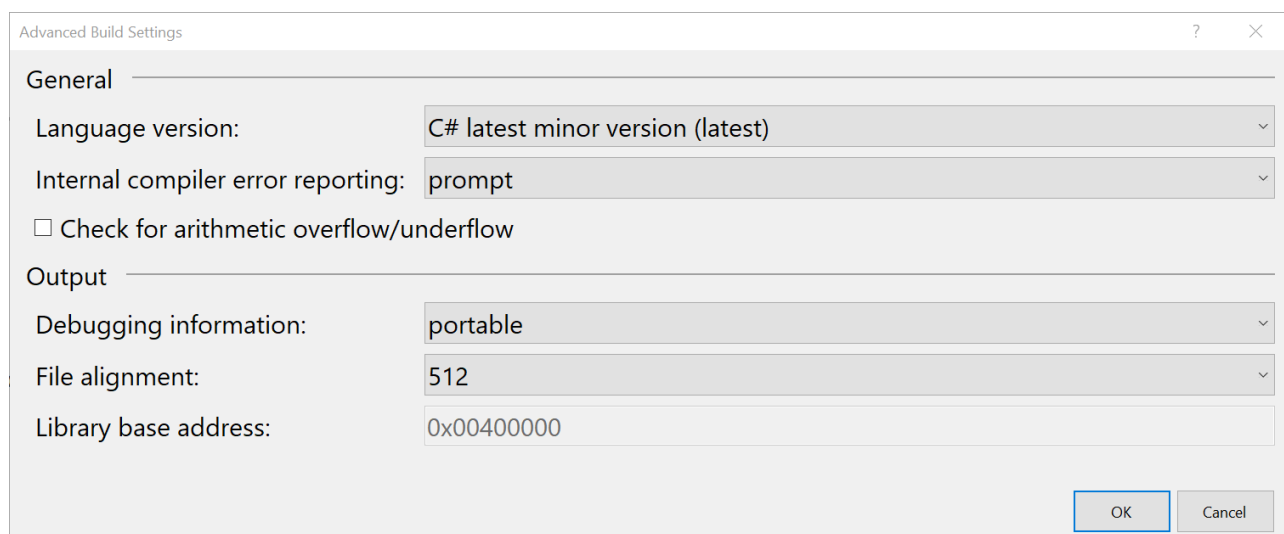
- Rely on a [Visual Studio quick action](#).
- Set the language version in the [Visual Studio UI](#).
- Manually edit your [.csproj](#) file.
- Set the language version [for multiple projects in a subdirectory](#).
- Configure the `-langversion` [compiler option](#).

Visual Studio quick action

Visual Studio helps you determine the language version you need. If you use a language feature that is not available for the currently configured version, Visual Studio shows a potential fix to update the language version for the project.

Set the language version in Visual Studio

You can set the version in Visual Studio. Right-click on the project node in Solution Explorer and select **Properties**. Select the **Build** tab and select the **Advanced** button. In the dropdown, select the version. The following image shows the "latest" setting:



NOTE

If you use the Visual Studio IDE to update your csproj files, the IDE creates separate nodes for each build configuration. You'll typically set the value the same in all build configurations, but you need to set it explicitly for each build configuration, or select "All Configurations" when you modify this setting. You'll see the following in your csproj file:

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
  <LangVersion>latest</LangVersion>
</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

Edit the csproj file

You can set the language version in your **.csproj** file. Add an element like the following:

```
<PropertyGroup>
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

The value `latest` uses the latest minor version of the C# language. Valid values are:

VALUE	MEANING
default	The compiler accepts all valid language syntax from the latest major version that it can support.
ISO-1	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2)
ISO-2	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0)
3	The compiler accepts only syntax that is included in C# 3.0 or lower.
4	The compiler accepts only syntax that is included in C# 4.0 or lower.
5	The compiler accepts only syntax that is included in C# 5.0 or lower.
6	The compiler accepts only syntax that is included in C# 6.0 or lower.
7	The compiler accepts only syntax that is included in C# 7.0 or lower.
7.1	The compiler accepts only syntax that is included in C# 7.1 or lower.
7.2	The compiler accepts only syntax that is included in C# 7.2 or lower.

VALUE	MEANING
7.3	The compiler accepts only syntax that is included in C# 7.3 or lower.
latest	The compiler accepts all valid language syntax that it can support.

The special strings `default` and `latest` resolve to the latest major (C# 7.0) and minor (C# 7.3) language versions installed on the build machine, respectively.

Configure multiple projects

You can create a **Directory.build.props** file that contains the `<LangVersion>` element to configure multiple directories. You typically do that in your solution directory. Add the following to a **Directory.build.props** file in your solution directory:

```
<Project>
  <PropertyGroup>
    <LangVersion>7.3</LangVersion>
  </PropertyGroup>
</Project>
```

Now, builds in every subdirectory of the directory containing that file will use C# version 7.3 syntax. For more information, see the article on [Customize your build](#).

Set the langversion compiler option

You can use the `-langversion` command-line option. For more information, see the article on the [-langversion](#) compiler option. You can see a list of the valid values by typing `csc -langversion:?`.

C# Keywords

5/4/2018 • 2 minutes to read • [Edit Online](#)

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a valid identifier, but `if` is not because `if` is a keyword.

The first table in this topic lists keywords that are reserved identifiers in any part of a C# program. The second table in this topic lists the contextual keywords in C#. Contextual keywords have special meaning only in a limited program context and can be used as identifiers outside that context. Generally, as new keywords are added to the C# language, they are added as contextual keywords in order to avoid breaking programs written in earlier versions.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort

<code>using</code>	<code>using static</code>	<code>virtual</code>	<code>void</code>
<code>volatile</code>	<code>while</code>		

Contextual Keywords

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. Some contextual keywords, such as `partial` and `where`, have special meanings in two or more contexts.

<code>add</code>	<code>alias</code>	<code>ascending</code>
<code>async</code>	<code>await</code>	<code>descending</code>
<code>dynamic</code>	<code>from</code>	<code>get</code>
<code>global</code>	<code>group</code>	<code>into</code>
<code>join</code>	<code>let</code>	<code>nameof</code>
<code>orderby</code>	<code>partial (type)</code>	<code>partial (method)</code>
<code>remove</code>	<code>select</code>	<code>set</code>
<code>value</code>	<code>var</code>	<code>when (filter condition)</code>
<code>where (generic type constraint)</code>	<code>where (query clause)</code>	<code>yield</code>

See Also

- [C# Reference](#)
- [C# Programming Guide](#)

C# Operators

5/22/2018 • 7 minutes to read • [Edit Online](#)

C# provides many operators, which are symbols that specify which operations (math, indexing, function call, etc.) to perform in an expression. You can [overload](#) many operators to change their meaning when applied to a user-defined type.

Operations on integral types (such as `==`, `!=`, `<`, `>`, `&`, `|`) are generally allowed on enumeration (`enum`) types.

The sections below list the C# operators starting with the highest precedence to the lowest. The operators within each section share the same precedence level.

Primary Operators

These are the highest precedence operators.

`x.y` – member access.

`x?.y` – null conditional member access. Returns `null` if the left-hand operand evaluates to `null`.

`x?[y]` – null conditional index access. Returns `null` if the left-hand operand evaluates to `null`.

`f(x)` – function invocation.

`a[x]` – aggregate object indexing.

`x++` – postfix increment. Returns the value of `x` and then updates the storage location with the value of `x` that is one greater (typically adds the integer 1).

`--x` – postfix decrement. Returns the value of `x` and then updates the storage location with the value of `x` that is one less (typically subtracts the integer 1).

`new` – type instantiation.

`typeof` – returns the `Type` object representing the operand.

`checked` – enables overflow checking for integer operations.

`unchecked` – disables overflow checking for integer operations. This is the default compiler behavior.

`default(T)` – produces the default value of type `T`.

`delegate` – declares and returns a delegate instance.

`sizeof` – returns the size in bytes of the type operand.

`->` – pointer dereferencing combined with member access.

Unary Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`+x` – returns the value of `x`.

`-x` – numeric negation.

`!x` – logical negation.

`~x` – bitwise complement.

`++x` – prefix increment. Returns the value of `x` after updating the storage location with the value of `x` that is one greater (typically adds the integer 1).

`--x` – prefix decrement. Returns the value of `x` after updating the storage location with the value of `x` that is one less (typically subtracts the integer 1).

`(T)x` – type casting.

`await` – awaits a `Task`.

`&x` – address of.

`*x` – dereferencing.

Multiplicative Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x * y` – multiplication.

`x / y` – division. If the operands are integers, the result is an integer truncated toward zero (for example, `-7 / 2` is `-3`).

`x % y` – remainder. If the operands are integers, this returns the remainder of dividing `x` by `y`. If `q = x / y` and `r = x % y`, then `x = q * y + r`.

Additive Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x + y` – addition.

`x - y` – subtraction.

Shift Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x << y` – shift bits left and fill with zero on the right.

`x >> y` – shift bits right. If the left operand is `int` or `long`, then left bits are filled with the sign bit. If the left operand is `uint` or `ulong`, then left bits are filled with zero.

Relational and Type-testing Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x < y` – less than (true if `x` is less than `y`).

`x > y` – greater than (true if `x` is greater than `y`).

`x <= y` – less than or equal to.

`x >= y` – greater than or equal to.

`is` – type compatibility. Returns true if the evaluated left operand can be cast to the type specified in the right operand (a static type).

`as` – type conversion. Returns the left operand cast to the type specified by the right operand (a static type), but `as`

returns `null` where `(T)x` would throw an exception.

Equality Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x == y` – equality. By default, for reference types other than `string`, this returns reference equality (identity test). However, types can overload `==`, so if your intent is to test identity, it is best to use the `ReferenceEquals` method on `object`.

`x != y` – not equal. See comment for `==`. If a type overloads `==`, then it must overload `!=`.

Logical AND Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x & y` – logical or bitwise AND. You can generally use this with integer types and `enum` types.

Logical XOR Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x ^ y` – logical or bitwise XOR. You can generally use this with integer types and `enum` types.

Logical OR Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x | y` – logical or bitwise OR. You can generally use this with integer types and `enum` types.

Conditional AND Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x && y` – logical AND. If the first operand evaluates to false, then C# does not evaluate the second operand.

Conditional OR Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x || y` – logical OR. If the first operand evaluates to true, then C# does not evaluate the second operand.

Null-coalescing Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`x ?? y` – returns `x` if it is non-`null`; otherwise, returns `y`.

Conditional Operator

This operator has higher precedence than the next section and lower precedence than the previous section.

`t ? x : y` – if test `t` evaluates to true, then evaluate and return `x`; otherwise, evaluate and return `y`.

Assignment and Lambda Operators

These operators have higher precedence than the next section and lower precedence than the previous section.

`x = y` – assignment.

`x += y` – increment. Add the value of `y` to the value of `x`, store the result in `x`, and return the new value. If `x` designates an `event`, then `y` must be an appropriate function that C# adds as an event handler.

`x -= y` – decrement. Subtract the value of `y` from the value of `x`, store the result in `x`, and return the new value. If `x` designates an `event`, then `y` must be an appropriate function that C# removes as an event handler.

`x *= y` – multiplication assignment. Multiply the value of `y` to the value of `x`, store the result in `x`, and return the new value.

`x /= y` – division assignment. Divide the value of `x` by the value of `y`, store the result in `x`, and return the new value.

`x %= y` – remainder assignment. Divide the value of `x` by the value of `y`, store the remainder in `x`, and return the new value.

`x &= y` – AND assignment. AND the value of `y` with the value of `x`, store the result in `x`, and return the new value.

`x |= y` – OR assignment. OR the value of `y` with the value of `x`, store the result in `x`, and return the new value.

`x ^= y` – XOR assignment. XOR the value of `y` with the value of `x`, store the result in `x`, and return the new value.

`x <<= y` – left-shift assignment. Shift the value of `x` left by `y` places, store the result in `x`, and return the new value.

`x >>= y` – right-shift assignment. Shift the value of `x` right by `y` places, store the result in `x`, and return the new value.

`=>` – lambda declaration.

Arithmetic Overflow

The arithmetic operators (`+`, `-`, `*`, `/`) can produce results that are outside the range of possible values for the numeric type involved. You should refer to the section on a particular operator for details, but in general:

- Integer arithmetic overflow either throws an [OverflowException](#) or discards the most significant bits of the result. Integer division by zero always throws a [DivideByZeroException](#).

When integer overflow occurs, what happens depends on the execution context, which can be [checked or unchecked](#). In a checked context, an [OverflowException](#) is thrown. In an unchecked context, the most significant bits of the result are discarded and execution continues. Thus, C# gives you the choice of handling or ignoring overflow. By default, arithmetic operations occur in an *unchecked* context.

In addition to the arithmetic operations, integral-type to integral-type casts can cause overflow (such as when you cast a [long](#) to an [int](#)), and are subject to checked or unchecked execution. However, bitwise operators and shift operators never cause overflow.

- Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).
- [Decimal](#) arithmetic overflow always throws an [OverflowException](#). Decimal division by zero always throws a [DivideByZeroException](#).

See Also

[C# Reference](#)

[C# Programming Guide](#)
[C# Overloadable Operators](#)
[C# Keywords](#)

C# Special Characters

5/4/2018 • 2 minutes to read • [Edit Online](#)

Special characters are predefined, contextual characters that modify the program element (a literal string, an identifier, or an attribute name) to which they are prepended. C# supports the following special characters:

- `@`, the verbatim identifier character.
- `$`, the interpolated string character.

See Also

[C# Reference](#)

[C# Programming Guide](#)

C# preprocessor directives

5/4/2018 • 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](#)

C# Compiler Options

5/24/2018 • 2 minutes to read • [Edit Online](#)

The compiler produces executable (.exe) files, dynamic-link libraries (.dll), or code modules (.netmodule).

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the **-option** form.

In Visual Studio, you set compiler options in the web.config file. For more information, see [<compiler> Element](#).

In This Section

[Command-line Building With csc.exe](#)

Information about building a Visual C# application from the command line.

[How to: Set Environment Variables for the Visual Studio Command Line](#)

Provides steps for running vsvars32.bat to enable command-line builds.

[C# Compiler Options Listed by Category](#)

A categorical listing of the compiler options.

[C# Compiler Options Listed Alphabetically](#)

An alphabetical listing of the compiler options.

Related Sections

[Build Page, Project Designer](#)

Setting properties that govern how your project is compiled, built, and debugged. Includes information about custom build steps in Visual C# projects.

[Default and Custom Builds](#)

Information on build types and configurations.

[Preparing and Managing Builds](#)

Procedures for building within the Visual Studio development environment.

C# Compiler Errors

5/4/2018 • 2 minutes to read • [Edit Online](#)

Some C# compiler errors have corresponding topics that explain why the error is generated, and, in some cases, how to fix the error. Use one of the following steps to see whether help is available for a particular error message.

- Find the error number (for example, CS0029) in the [Output Window](#), and then search for it on Microsoft Docs.
- Choose the error number (for example, CS0029) in the [Output Window](#), and then choose the F1 key.
- In the Index, enter the error number in the **Look for** box.

If none of these steps leads to information about your error, go to the end of this page, and send feedback that includes the number or text of the error.

For information about how to configure error and warning options in C#, see [Build Page, Project Designer \(C#\)](#).

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

See Also

[C# Compiler Options](#)

[Sorry, we don't have specifics on this C# error](#)

[Build Page, Project Designer \(C#\)](#)

[/warn \(C# Compiler Options\)](#)

[/nowarn \(C# Compiler Options\)](#)

C# 6.0 draft language specification

5/24/2018 • 2 minutes to read • [Edit Online](#)

The C# language specification is the definitive source for C# syntax and usage. This specification contains detailed information about all aspects of the language, including many points that the documentation for C# doesn't cover.

Version 5.0 of the specification has been released in December 2017 as the [Standard ECMA-334 5th Edition](#) document.

Version 6.0 of the specification has not been approved as a standard. This site contains the [draft C# 6.0 specification](#). It's built from the markdown files contained in the [dotnet/csharplang GitHub repository](#).

Issues on the draft specification should be created in the [dotnet/csharplang](#) repository. Or, if you are interested in fixing any errors you find, you may submit a [Pull Request](#) to the same repository.

See also

[C# Reference](#)

[C# Programming Guide](#)

NEXT