Contents

-moduleassemblyname
-noconfig
-nologo
-nostdlib
-nowarn
-nowin32manifest
-optimize
-out
-pathmap
-pdb
-platform
-preferreduilang
-publicsign
-recurse
-reference
-refout
-refonly
-resource
-subsystemversion
-target
  -target:appcontainerexe
  -target:exe
  -target:library
  -target:module
  -target:winexe
  -target:winmdobj
-unsafe
-utf8output
-warn
-warnaserror
-win32icon
-win32manifest

-win32res

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.

Command-line build with csc.exe

1/23/2019 • 3 minutes to read • Edit Online

You can invoke the C# compiler by typing the name of its executable file (*csc.exe*) at a command prompt.

If you use the **Developer Command Prompt for Visual Studio** window, all the necessary environment variables are set for you. For information on how to access this tool, see the Developer Command Prompt for Visual Studio topic.

If you use a standard Command Prompt window, you must adjust your path before you can invoke *csc.exe* from any subdirectory on your computer. You also must run *vsvars32.bat* to set the appropriate environment variables to support command-line builds. For more information about *vsvars32.bat*, including instructions for how to find and run it, see How to: Set Environment Variables for the Visual Studio Command Line.

If you're working on a computer that has only the Windows Software Development Kit (SDK), you can use the C# compiler at the **SDK Command Prompt**, which you open from the **Microsoft .NET Framework SDK** menu option.

You can also use MSBuild to build C# programs programmatically. For more information, see MSBuild.

The *csc.exe* executable file usually is located in the Microsoft.NET\Framework\<*Version*> folder under the *Windows* directory. Its location might vary depending on the exact configuration of a particular computer. If more than one version of the .NET Framework is installed on your computer, you'll find multiple versions of this file. For more information about such installations, see How to: determine which versions of the .NET Framework are installed.

> **TIP**
> When you build a project by using the Visual Studio IDE, you can display the **csc** command and its associated compiler options in the **Output** window. To display this information, follow the instructions in How to: View, Save, and Configure Build Log Files to change the verbosity level of the log data to **Normal** or **Detailed**. After you rebuild your project, search the **Output** window for **csc** to find the invocation of the C# compiler.

In this topic

Rules for command-line syntax for the C# compiler

The C# compiler uses the following rules when it interprets arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled by the command-line parser in the operating system before it's passed to the `argv` array in the program.
- A string enclosed in double quotation marks ("string") is interpreted as a single argument, regardless of white space that is contained within. A quoted string can be embedded in an argument.

- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one backslash is put in the `argv` array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one backslash is put in the `argv` array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash. This causes a literal double quotation mark (") to be added in `argv`.

Sample command lines for the C# compiler

- Compiles *File.cs* producing *File.exe*:

```
csc File.cs
```

- Compiles *File.cs* producing *File.dll*:

```
csc -target:library File.cs
```

- Compiles *File.cs* and creates *My.exe*:

```
csc -out:My.exe File.cs
```

- Compiles all the C# files in the current directory with optimizations enabled and defines the DEBUG symbol. The output is *File2.exe*:

```
csc -define:DEBUG -optimize -out:File2.exe *.cs
```

- Compiles all the C# files in the current directory producing a debug version of *File2.dll*. No logo and no warnings are displayed:

```
csc -target:library -out:File2.dll -warn:0 -nologo -debug *.cs
```

- Compiles all the C# files in the current directory to *Something.xyz* (a DLL):

```
csc -target:library -out:Something.xyz *.cs
```

Differences between C# compiler and C++ compiler output

There are no object (*.obj*) files created as a result of invoking the C# compiler; output files are created directly. As a result of this, the C# compiler does not need a linker.

See also

- C# Compiler Options
- C# Compiler Options Listed Alphabetically
- C# Compiler Options Listed by Category

- Main() and Command-Line Arguments
- Command-Line Arguments
- How to: Display Command-Line Arguments
- How to: Access Command-Line Arguments Using foreach
- Main() Return Values

How to: Set Environment Variables for the Visual Studio Command Line

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

The VsDevCmd.bat file sets the appropriate environment variables to enable command-line builds.

> **NOTE**
>
> The VsDevCmd.bat file is a new file delivered with Visual Studio 2017. Visual Studio 2015 and earlier versions used VSVARS32.bat for the same purpose. This file was stored in \Program Files\Microsoft Visual Studio*Version*\Common7\Tools or Program Files (x86)\Microsoft Visual Studio*Version*\Common7\Tools.

If the current version of Visual Studio is installed on a computer that also has an earlier version of Visual Studio, you should not run VsDevCmd.bat and VSVARS32.BAT from different versions in the same Command Prompt window. Instead, you should run the command for each version in its own window.

To run VsDevCmd.BAT

1. From the **Start** menu, open the **Developer Command Prompt for VS 2017**. It's in the **Visual Studio 2017** folder.
2. Change to the \Program Files\Microsoft Visual Studio*Version**Offering*\Common7\Tools or \Program Files (x86)\Microsoft Visual Studio*Version**Offering*\Common7\Tools subdirectory of your installation. (*Version* is *2017* for the current version. *Offering* is one of *Enterprise*, *Professional* or *Community*.)
3. Run VsDevCmd.bat by typing **VsDevCmd**.

   Caution

   VsDevCmd.bat can vary from computer to computer. Do not replace a missing or damaged VsDevCmd.bat file with a VsDevCmd.bat from another computer. Instead, rerun setup to replace the missing file.

Available options for VsDevCmd.BAT

To see the available options for VsDevCmd.BAT, run the command with the `-help` option:

```
VsDevCmd.bat -help
```

See also

- Command-line Building With csc.exe

C# Compiler Options Listed by Category

1/23/2019 • 3 minutes to read • Edit Online

The following compiler options are sorted by category. For an alphabetical list, see C# Compiler Options Listed Alphabetically.

Optimization

OPTION	PURPOSE
-filealign	Specifies the size of sections in the output file.
-optimize	Enables/disables optimizations.

Output Files

OPTION	PURPOSE
-deterministic	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
-doc	Specifies an XML file where processed documentation comments are to be written.
-out	Specifies the output file.
-pathmap	Specify a mapping for source path names output by the compiler
-pdb	Specifies the file name and location of the .pdb file.
-platform	Specify the output platform.
-preferreduilang	Specify a language for compiler output.
-refout	Generate a reference assembly in addition to the primary assembly.
-refonly	Generate a reference assembly instead of a primary assembly.
-target	Specifies the format of the output file using one of five options: -target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe, or -target:winmdobj.
-modulename:<string>	Specify the name of the source module

.NET Framework Assemblies

OPTION	PURPOSE
-addmodule	Specifies one or more modules to be part of this assembly.
-delaysign	Instructs the compiler to add the public key but to leave the assembly unsigned.
-keycontainer	Specifies the name of the cryptographic key container.
-keyfile	Specifies the filename containing the cryptographic key.
-lib	Specifies the location of assemblies referenced by means of -reference.
-nostdlib	Instructs the compiler not to import the standard library (mscorlib.dll).
-publicsign	Apply a public key without signing the assembly, but set the bit in the assembly indicating the assembly is signed.
-reference	Imports metadata from a file that contains an assembly.
-analyzer	Run the analyzers from this assembly (Short form: /a)
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.

Debugging/Error Checking

OPTION	PURPOSE
-bugreport	Creates a file that contains information that makes it easy to report a bug.
-checked	Specifies whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
-debug	Instruct the compiler to emit debugging information.
-errorreport	Sets error reporting behavior.
-fullpaths	Specifies the absolute path to the file in compiler output.
-nowarn	Suppresses the compiler's generation of specified warnings.
-warn	Sets the warning level.
-warnaserror	Promotes warnings to errors.
-ruleset:<file>	Specify a ruleset file that disables specific diagnostics.

Preprocessor

OPTION	PURPOSE
-define	Defines preprocessor symbols.

Resources

OPTION	PURPOSE
-link	Makes COM type information in specified assemblies available to the project.
-linkresource	Creates a link to a managed resource.
-resource	Embeds a .NET Framework resource into the output file.
-win32icon	Specifies an .ico file to insert into the output file.
-win32res	Specifies a Win32 resource to insert into the output file.

Miscellaneous

OPTION	PURPOSE
@	Specifies a response file.
-?	Lists compiler options to stdout.
-baseaddress	Specifies the preferred base address at which to load a DLL.
-codepage	Specifies the code page to use for all source code files in the compilation.
-help	Lists compiler options to stdout.
-highentropyva	Specifies that the executable file supports address space layout randomization (ASLR).
-langversion	Specify language version: Default, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3, or Latest
-main	Specifies the location of the **Main** method.
-noconfig	Instructs the compiler not to compile with csc.rsp.
-nologo	Suppresses compiler banner information.
-recurse	Searches subdirectories for source files to compile.
-subsystemversion	Specifies the minimum version of the subsystem that the executable file can use.

OPTION	PURPOSE
-unsafe	Enables compilation of code that uses the unsafe keyword.
-utf8output	Displays compiler output using UTF-8 encoding.
-parallel[+|-]	Specifies whether to use concurrent build (+).
-checksumalgorithm:<alg>	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.

Obsolete Options

OPTION	PURPOSE
-incremental	Enables incremental compilation.

See also

- C# Compiler Options
- C# Compiler Options Listed Alphabetically
- How to: Set Environment Variables for the Visual Studio Command Line

C# Compiler Options Listed Alphabetically

1/23/2019 • 3 minutes to read • Edit Online

The following compiler options are sorted alphabetically. For a categorical list, see C# Compiler Options Listed by Category.

OPTION	PURPOSE
@	Reads a response file for more options.
-?	Displays a usage message to stdout.
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.
-addmodule	Links the specified modules into this assembly
-analyzer	Run the analyzers from this assembly (Short form: -a)
-appconfig	Specifies the location of app.config at assembly binding time.
-baseaddress	Specifies the base address for the library to be built.
-bugreport	Creates a 'Bug Report' file. This file will be sent together with any crash information if it is used with -errorreport:prompt or -errorreport:send.
-checked	Causes the compiler to generate overflow checks.
-checksumalgorithm:<alg>	Specifies the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.
-codepage	Specifies the codepage to use when opening source files.
-debug	Emits debugging information.
-define	Defines conditional compilation symbols.
-delaysign	Delay-signs the assembly by using only the public part of the strong name key.
-deterministic	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
-doc	Specifies an XML Documentation file to generate.
-errorreport	Specifies how to handle internal compiler errors: prompt, send, or none. The default is none.

OPTION	PURPOSE
-filealign	Specifies the alignment used for output file sections.
-fullpaths	Causes the compiler to generate fully qualified paths.
-help	Displays a usage message to stdout.
-highentropyva	Specifies that high entropy ASLR is supported.
-incremental	Enables incremental compilation [obsolete].
-keycontainer	Specifies a strong name key container.
-keyfile	Specifies a strong name key file.
-langversion:<string>	Specify language version: Default, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3, or Latest
-lib	Specifies additional directories in which to search for references.
-link	Makes COM type information in specified assemblies available to the project.
-linkresource	Links the specified resource to this assembly.
-main	Specifies the type that contains the entry point (ignore all other possible entry points).
-moduleassemblyname	Specifies an assembly whose non-public types a .netmodule can access.
-modulename:<string>	Specify the name of the source module
-noconfig	Instructs the compiler not to auto include CSC.RSP file.
-nologo	Suppresses compiler copyright message.
-nostdlib	Instructs the compiler not to reference standard library (mscorlib.dll).
-nowarn	Disables specific warning messages
-nowin32manifest	Instructs the compiler not to embed an application manifest in the executable file.
-optimize	Enables/disables optimizations.
-out	Specifies the output file name (default: base name of file with main class or first file).
-parallel[+|-]	Specifies whether to use concurrent build (+).

OPTION	PURPOSE
-pathmap	Specifies a mapping for source path names output by the compiler.
-pdb	Specifies the file name and location of the .pdb file.
-platform	Limits which platforms this code can run on: x86, Itanium, x64, anycpu, or anycpu32bitpreferred. The default is anycpu.
-preferreduilang	Specifies the language to be used for compiler output.
-publicsign	Apply a public key without signing the assembly, but set the bit in the assembly indicating the assembly is signed.
-recurse	Includes all files in the current directory and subdirectories according to the wildcard specifications.
-reference	References metadata from the specified assembly files.
-refout	Generate a reference assembly in addition to the primary assembly.
-refonly	Generate a reference assembly instead of a primary assembly.
-resource	Embeds the specified resource.
-ruleset:<file>	Specify a ruleset file that disables specific diagnostics.
-subsystemversion	Specifies the minimum version of the subsystem that the executable file can use.
-target	Specifies the format of the output file by using one of four options: -target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe, -target:winmdobj.
-unsafe	Allows unsafe code.
-utf8output	Outputs compiler messages in UTF-8 encoding.
-warn	Sets the warning level (0-4).
-warnaserror	Reports specific warnings as errors.
-win32icon	Uses this icon for the output.
-win32manifest	Specifies a custom win32 manifest file.
-win32res	Specifies the win32 resource file (.res).

See also

- C# Compiler Options
- C# Compiler Options Listed by Category

- How to: Set Environment Variables for the Visual Studio Command Line
- <compiler> Element

@ (C# Compiler Options)

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

The @ option lets you specify a file that contains compiler options and source code files to compile.

Syntax

```
@response_file
```

Arguments

`response_file`
A file that lists compiler options or source code files to compile.

Remarks

The compiler options and source code files will be processed by the compiler just as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response file options. For example:

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the # symbol.

Specifying compiler options from within a response file is just like issuing those commands on the command line. See Building from the Command Line for more information.

The compiler processes the command options as they are encountered. Therefore, command line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files.

C# provides the csc.rsp file, which is located in the same directory as the csc.exe file. See -noconfig for more information on csc.rsp.

This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

Example

The following are a few lines from a sample response file:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

See also

- C# Compiler Options

-addmodule (C# Compiler Options)

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

This option adds a module that was created with the target:module switch to the current compilation.

Syntax

```
-addmodule:file[;file2]
```

Arguments

`file`, `file2`
An output file that contains metadata. The file cannot contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

Remarks

All modules added with **-addmodule** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module is not in the application directory at run time, you will get a TypeLoadException.

`file` cannot contain an assembly. For example, if the output file was created with -target:module, its metadata can be imported with **-addmodule**.

If the output file was created with a **-target** option other than **-target:module**, its metadata cannot be imported with **-addmodule** but can be imported with -reference.

This compiler option is unavailable in Visual Studio; a project cannot reference a module. In addition, this compiler option cannot be changed programmatically.

Example

Compile source file `input.cs` and add metadata from `metad1.netmodule` and `metad2.netmodule` to produce `out.exe`:

```
csc -addmodule:metad1.netmodule;metad2.netmodule -out:out.exe input.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties
- Multifile Assemblies
- How to: Build a Multifile Assembly

-appconfig (C# Compiler Options)

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

The **-appconfig** compiler option enables a C# application to specify the location of an assembly's application configuration (app.config) file to the common language runtime (CLR) at assembly binding time.

Syntax

```
-appconfig:file
```

Arguments

`file`

Required. The application configuration file that contains assembly binding settings.

Remarks

One use of **-appconfig** is advanced scenarios in which an assembly has to reference both the .NET Framework version and the .NET Framework for Silverlight version of a particular reference assembly at the same time. For example, a XAML designer written in Windows Presentation Foundation (WPF) might have to reference both the WPF Desktop, for the designer's user interface, and the subset of WPF that is included with Silverlight. The same designer assembly has to access both assemblies. By default, the separate references cause a compiler error, because assembly binding sees the two assemblies as equivalent.

The **-appconfig** compiler option enables you to specify the location of an app.config file that disables the default behavior by using a `<supportPortability>` tag, as shown in the following example.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

The compiler passes the location of the file to the CLR's assembly-binding logic.

> **NOTE**
>
> If you are using the Microsoft Build Engine (MSBuild) to build your application, you can set the **-appconfig** compiler option by adding a property tag to the .csproj file. To use the app.config file that is already set in the project, add property tag `<UseAppConfigForCompiler>` to the .csproj file and set its value to `true`. To specify a different app.config file, add property tag `<AppConfigForCompiler>` and set its value to the location of the file.

Example

The following example shows an app.config file that enables an application to have references to both the .NET Framework implementation and the .NET Framework for Silverlight implementation of any .NET Framework assembly that exists in both implementations. The **-appconfig** compiler option specifies the location of this app.config file.

```
<configuration>
      <runtime>
      <assemblyBinding>
            <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
            <supportPortability PKT="31bf3856ad364e35" enable="false"/>
      </assemblyBinding>
      </runtime>
</configuration>
```

See also

- <supportPortability> Element
- C# Compiler Options Listed Alphabetically

-baseaddress (C# Compiler Options)

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

The **-baseaddress** option lets you specify the preferred base address at which to load a DLL. For more information about when and why to use this option, see Larry Osterman's WebLog.

Syntax

```
-baseaddress:address
```

Arguments

`address`

The base address for the DLL. This address can be specified as a decimal, hexadecimal, or octal number.

Remarks

The default base address for a DLL is set by the .NET Framework common language runtime.

Be aware that the lower-order word in this address will be rounded. For example, if you specify 0x11110001, it will be rounded to 0x11110000.

To complete the signing process for a DLL, use SN.EXE with the -R option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Click the **Advanced** button.
4. Modify the **DLL Base Address** property.

   To set this compiler option programmatically, see BaseAddress.

See also

- ProcessModule.BaseAddress
- C# Compiler Options
- Managing Project and Solution Properties

-bugreport (C# Compiler Options)

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

Specifies that debug information should be placed in a file for later analysis.

Syntax

```
-bugreport:file
```

Arguments

`file`
The name of the file that you want to contain your bug report.

Remarks

The **-bugreport** option specifies that the following information should be placed in `file`:

- A copy of all source code files in the compilation.
- A listing of the compiler options used in the compilation.
- Version information about your compiler, run time, and operating system.
- Referenced assemblies and modules, saved as hexadecimal digits, except assemblies that ship with the .NET Framework and SDK.
- Compiler output, if any.
- A description of the problem, which you will be prompted for.
- A description of how you think the problem should be fixed, which you will be prompted for.

If this option is used with **-errorreport:prompt** or **-errorreport:send**, the information in the file will be sent to Microsoft Corporation.

Because a copy of all source code files will be placed in `file`, you might want to reproduce the suspected code defect in the shortest possible program.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Notice that contents of the generated file expose source code that could result in inadvertent information disclosure.

See also

- C# Compiler Options
- -errorreport (C# Compiler Options)
- Managing Project and Solution Properties

-checked (C# Compiler Options)

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

The **-checked** option specifies whether an integer arithmetic statement that results in a value that is outside the range of the data type, and that is not in the scope of a checked or unchecked keyword, causes a run-time exception.

Syntax

```
-checked[+ | -]
```

Remarks

An integer arithmetic statement that is in the scope of a `checked` or `unchecked` keyword is not subject to the effect of the **-checked** option.

If an integer arithmetic statement that is not in the scope of a `checked` or `unchecked` keyword results in a value outside the range of the data type, and **-checked+** (or **-checked**) is used in the compilation, that statement causes an exception at run time. If **-checked-** is used in the compilation, that statement does not cause an exception at run time.

The default value for this option is **-checked-**; overflow checking is disabled.

Sometimes, automated tools that are used to build large applications set -checked to +. One scenario for using -checked- is to override the tool's global default by specifying -checked-.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page. For more information, see Build Page, Project Designer (C#).
2. Click the **Build** property page.
3. Click the **Advanced** button.
4. Modify the **Check for arithmetic overflow/underflow** property.

To access this compiler option programmatically, see CheckForOverflowUnderflow.

Example

The following command compiles `t2.cs`. The use of `-checked` in the command specifies that any integer arithmetic statement in the file that is not in the scope of a `checked` or `unchecked` keyword, and that results in a value that is outside the range of the data type, causes an exception at run time.

```
csc t2.cs -checked
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-codepage (C# Compiler Options)

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

This option specifies which codepage to use during compilation if the required page is not the current default codepage for the system.

Syntax

```
-codepage:id
```

Arguments

`id`
The id of the code page to use for all source code files in the compilation.

Remarks

If you compile one or more source code files that were not created to use the default code page on your computer, you can use the **-codepage** option to specify which code page should be used. **-codepage** applies to all source code files in your compilation.

If the source code files were created with the same codepage that is in effect on your computer or if the source code files were created with UNICODE or UTF-8, you need not use **-codepage**.

See GetCPInfo for information on how to find which code pages are supported on your system.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-debug (C# Compiler Options)

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

The **-debug** option causes the compiler to generate debugging information and place it in the output file or files.

Syntax

```
-debug[+ | -]
-debug:{full | pdbonly}
```

Arguments

`+` | `-`

Specifying `+`, or just **-debug**, causes the compiler to generate debugging information and place it in a program database (.pdb file). Specifying `-`, which is in effect if you do not specify **-debug**, causes no debug information to be created.

`full` | `pdbonly`

Specifies the type of debugging information generated by the compiler. The full argument, which is in effect if you do not specify **-debug:pdbonly**, enables attaching a debugger to the running program. Specifying pdbonly allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger.

Remarks

Use this option to create debug builds. If **-debug**, **-debug+**, or **-debug:full** is not specified, you will not be able to debug the output file of your program.

If you use **-debug:full**, be aware that there is some impact on the speed and size of JIT optimized code and a small impact on code quality with **-debug:full**. We recommend **-debug:pdbonly** or no PDB for generating release code.

> **NOTE**
> One difference between **-debug:pdbonly** and **-debug:full** is that with **-debug:full** the compiler emits a DebuggableAttribute, which is used to tell the JIT compiler that debug information is available. Therefore, you will get an error if your code contains the DebuggableAttribute set to false if you use **-debug:full**.

For more information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

To change the location of the .pdb file, see -pdb (C# Compiler Options).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Click the **Advanced** button.
4. Modify the **Debug Info** property.

For information on how to set this compiler option programmatically, see DebugSymbols.

Example

Place debugging information in output file `app.pdb`:

```
csc -debug -pdb:app.pdb test.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-define (C# Compiler Options)

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

The **-define** option defines `name` as a symbol in all source code files your program.

Syntax

```
-define:name[;name2]
```

Arguments

`name`, `name2`
The name of one or more symbols that you want to define.

Remarks

The **-define** option has the same effect as using a #define preprocessor directive except that the compiler option is in effect for all files in the project. A symbol remains defined in a source file until an #undef directive in the source file removes the definition. When you use the -define option, an `#undef` directive in one file has no effect on other source code files in the project.

You can use symbols created by this option with #if, #else, #elif, and #endif to compile source files conditionally.

-d is the short form of **-define**.

You can define multiple symbols with **-define** by using a semicolon or comma to separate symbol names. For example:

```
-define:DEBUG;TUESDAY
```

The C# compiler itself defines no symbols or macros that you can use in your source code; all symbol definitions must be user-defined.

> **NOTE**
> The C# `#define` does not allow a symbol to be given a value, as in languages such as C++. For example, `#define` cannot be used to create a macro or to define a constant. If you need to define a constant, use an `enum` variable. If you want to create a C++ style macro, consider alternatives such as generics. Since macros are notoriously error-prone, C# disallows their use but provides safer alternatives.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. On the **Build** tab, type the symbol that is to be defined in the **Conditional compilation symbols** box. For example, if you are using the code example that follows, just type `xx` into the text box.

For information on how to set this compiler option programmatically, see DefineConstants.

Example

```
// preprocessor_define.cs
// compile with: -define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-delaysign (C# Compiler Options)

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

This option causes the compiler to reserve space in the output file so that a digital signature can be added later.

Syntax

```
-delaysign[ + | - ]
```

Arguments

`+` | `-`

Use **-delaysign-** if you want a fully signed assembly. Use **-delaysign+** if you only want to place the public key in the assembly. The default is **-delaysign-**.

Remarks

The **-delaysign** option has no effect unless used with -keyfile or -keycontainer.

The **-delaysign** and **-publicsign** options are mutually exclusive.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. That operation creates a digital signature which is stored in the file that contains the manifest. When an assembly is delay signed, the compiler does not compute and store the signature, but reserves space in the file so the signature can be added later.

For example, using **-delaysign+** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly using the Assembly Linker utility.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment

1. Open the **Properties** page for the project.
2. Modify the **Delay sign only** property.

For information on how to set this compiler option programmatically, see DelaySign.

See also

- C# -publicsign option
- C# Compiler Options
- Managing Project and Solution Properties

-deterministic

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

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

Syntax

```
-deterministic
```

Remarks

By default, compiler output from a given set of inputs is unique, since the compiler adds a timestamp and a GUID that is generated from random numbers. You use the `-deterministic` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same.

The compiler considers the following inputs for the purpose of determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
  - Source files
  - Referenced assemblies
  - Referenced modules
  - Resources
  - The strong name key file
  - @ response files
  - Analyzers
  - Rulesets
  - Additional files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding is not specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `/lib` or `/recurse`).
- The CLR platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

When sources are publicly available, deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. It can also be useful in a continuous build system for determining whether build steps that are dependent on changes to a binary need to be executed.

See also

- C# Compiler Options

- Managing Project and Solution Properties

-doc (C# Compiler Options)

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

The **-doc** option allows you to place documentation comments in an XML file.

Syntax

```
-doc:file
```

Arguments

`file`

The output file for XML, which is populated with the comments in the source code files of the compilation.

Remarks

In source code files, documentation comments that precede the following can be processed and added to the XML file:

- Such user-defined types as a class, delegate, or interface
- Such members as a field, event, property, or method

The source code file that contains Main is output first into the XML.

To use the generated .xml file for use with the IntelliSense feature, let the file name of the .xml file be the same as the assembly you want to support and then make sure the .xml file is in the same directory as the assembly. Thus, when the assembly is referenced in the Visual Studio project, the .xml file is found as well. See Supplying Code Comments and for more information.

Unless you compile with -target:module, `file` will contain <assembly></assembly> tags specifying the name of the file containing the assembly manifest for the output file of the compilation.

> **NOTE**
> The -doc option applies to all input files; or, if set in the Project Settings, all files in the project. To disable warnings related to documentation comments for a specific file or section of code, use #pragma warning.

See Recommended Tags for Documentation Comments for ways to generate documentation from comments in your code.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** tab.
3. Modify the **XML documentation file** property.

For information on how to set this compiler option programmatically, see DocumentationFile.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-errorreport (C# Compiler Options)

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

This option provides a convenient way to report a C# internal compiler error to Microsoft.

> **NOTE**
> On Windows Vista and Windows Server 2008, the error reporting settings that you make for Visual Studio do not override the settings made through Windows Error Reporting (WER). WER settings always take precedence over Visual Studio error reporting settings.

Syntax

```
-errorreport:{ none | prompt | queue | send }
```

Arguments

none
Reports about internal compiler errors will not be collected or sent to Microsoft.

prompt
Prompts you to send a report when you receive an internal compiler error. **prompt** is the default when you compile an application in the development environment.

queue
Queues the error report. When you log on with administrative credentials, you can report any failures since the last time that you were logged on. You will not be prompted to send reports for failures more than once every three days. **queue** is the default when you compile an application at the command line.

send
Automatically sends reports of internal compiler errors to Microsoft. To enable this option, you must first agree to the Microsoft data collection policy. The first time that you specify **-errorreport:send** on a computer, a compiler message will refer you to a Web site that contains the Microsoft data collection policy.

Remarks

An internal compiler error (ICE) results when the compiler cannot process a source code file. When an ICE occurs, the compiler does not produce an output file or any useful diagnostic that you can use to fix your code.

In previous releases, when you received an ICE, you were encouraged to contact Microsoft Product Support Services to report the problem. By using **-errorreport**, you can provide ICE information to the Visual C# team. Your error reports can help improve future compiler releases.

A user's ability to send reports depends on computer and user policy permissions.

For more information about error debugger, see Description of the Dr. Watson for Windows (Drwtsn32.exe) Tool.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page. For more information, see Build Page, Project Designer (C#).
2. Click the **Build** property page.

3. Click the **Advanced** button.

4. Modify the **Internal Compiler Error Reporting** property.

For information about how to set this compiler option programmatically, see ErrorReport.

See also

- C# Compiler Options

-filealign (C# Compiler Options)

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

The **-filealign** option lets you specify the size of sections in your output file.

Syntax

```
-filealign:number
```

Arguments

`number`

A value that specifies the size of sections in the output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

Remarks

Each section will be aligned on a boundary that is a multiple of the **-filealign** value. There is no fixed default. If **-filealign** is not specified, the common language runtime picks a default at compile time.

By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that will run on smaller devices.

Use DUMPBIN to see information about sections in your output file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Click the **Advanced** button.
4. Modify the **File Alignment** property.

For information on how to set this compiler option programmatically, see FileAlignment.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-fullpaths (C# Compiler Options)

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

The **-fullpaths** option causes the compiler to specify the full path to the file when listing compilation errors and warnings.

Syntax

```
-fullpaths
```

Remarks

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The **-fullpaths** option causes the compiler to specify the full path to the file.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options

-help, -? (C# Compiler Options)

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

This option sends a listing of compiler options, and a brief description of each option, to stdout.

Syntax

```
-help
-?
```

Remarks

If this option is included in a compilation, no output file will be created and no compilation will take place.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-highentropyva (C# Compiler Options)

5/4/2018 • 2 minutes to read • Edit Online

The **-highentropyva** compiler option tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).

Syntax

```
-highentropyva[+ | -]
```

Arguments

`+` | `-`

This option specifies that a 64-bit executable or an executable that is marked by the -platform:anycpu compiler option supports a high entropy virtual address space. The option is disabled by default. Use **-highentropyva+** or **-highentropyva** to enable it.

Remarks

The **-highentropyva** option enables compatible versions of the Windows kernel to use higher degrees of entropy when randomizing the address space layout of a process as part of ASLR. Using higher degrees of entropy means that a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it is more difficult to guess the location of a particular memory region.

When the **-highentropyva** compiler option is specified, the target executable and any modules that it depends on must be able to handle pointer values that are larger than 4 gigabytes (GB) when they are running as a 64-bit process.

-keycontainer (C# Compiler Options)

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

Specifies the name of the cryptographic key container.

Syntax

```
-keycontainer:string
```

Arguments

`string`
The name of the strong name key container.

Remarks

When the **-keycontainer** option is used, the compiler creates a sharable component. The compiler inserts a public key from the specified container into the assembly manifest and signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. `sn -i` installs the key pair into a container. This option is not supported when the compiler runs on CoreCLR. To sign an assembly when building on CoreCLR, use the -keyfile option.

If you compile with -target:module, the name of the key file is held in the module and incorporated into the assembly when you compile this module into an assembly with -addmodule.

You can also specify this option as a custom attribute (System.Reflection.AssemblyKeyNameAttribute) in the source code for any Microsoft intermediate language (MSIL) module.

You can also pass your encryption information to the compiler with -keyfile. Use -delaysign if you want the public key added to the assembly manifest but want to delay signing the assembly until it has been tested.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment

1. This compiler option is not available in the Visual Studio development environment.

You can programmatically access this compiler option with AssemblyKeyContainerName.

See also

- C# Compiler -keyfile option
- C# Compiler Options
- Managing Project and Solution Properties

-keyfile (C# Compiler Options)

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

Specifies the filename containing the cryptographic key.

Syntax

```
-keyfile:file
```

Arguments

TERM	DEFINITION
`file`	The name of the file containing the strong name key.

Remarks

When this option is used, the compiler inserts the public key from the specified file into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type sn -k `file` at the command line.

If you compile with **-target:module**, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with -addmodule.

You can also pass your encryption information to the compiler with -keycontainer. Use -delaysign if you want a partially signed assembly.

In case both -keyfile and -keycontainer are specified (either by command line option or by custom attribute) in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler does not find the key container, it will try the file specified with -keyfile. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container (similar to sn -i) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment

1. Open the **Properties** page for the project.
2. Click the **Signing** property page.
3. Modify the **Choose a strong name key file** property.

You can programmatically access this compiler option with AssemblyOriginatorKeyFile.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-langversion (C# Compiler Options)

1/23/2019 • 3 minutes to read • Edit Online

Causes the compiler to accept only syntax that is included in the chosen C# language specification.

Syntax

```
-langversion:option
```

Arguments

`option`

The following values are valid:

OPTION	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) ISO1
ISO-2	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0) ISO2
3	The compiler accepts only syntax that is included in C# 3.0 or lower CS3
4	The compiler accepts only syntax that is included in C# 4.0 or lower CS4
5	The compiler accepts only syntax that is included in C# 5.0 or lower CS5
6	The compiler accepts only syntax that is included in C# 6.0 or lower CS6
7	The compiler accepts only syntax that is included in C# 7.0 or lower CS7
7.1	The compiler accepts only syntax that is included in C# 7.1 or lower CS71
7.2	The compiler accepts only syntax that is included in C# 7.2 or lower CS72
7.3	The compiler accepts only syntax that is included in C# 7.3 or lower CS73

OPTION	MEANING
latest	The compiler accepts all valid language syntax that it can support.

Remarks

Metadata referenced by your C# application is not subject to **-langversion** compiler option.

Because each version of the C# compiler contains extensions to the language specification, **-langversion** does not give you the equivalent functionality of an earlier version of the compiler.

Additionally, while C# version updates generally coincide with major .NET Framework releases, the new syntax and features are not necessarily tied to that specific framework version. While the new features definitely require a new compiler update that is also released alongside the C# revision, each specific feature has its own minimum .NET API or common language runtime requirements that may allow it to run on downlevel frameworks by including NuGet packages or other libraries.

Regardless of which **-langversion** setting you use, you will use the current version of the common language runtime to create your .exe or .dll. One exception is friend assemblies and -moduleassemblyname (C# Compiler Option), which work under **-langversion:ISO-1**.

For other ways to specify the C# language version, see the Select the C# language version topic.

For information about how to set this compiler option programmatically, see LanguageVersion.

See also

- C# Compiler Options
- Managing Project and Solution Properties

C# Language Specification

VERSION	LINK	DESCRIPTION
C# 1.0	Download DOC	C# Language Specification Version 1.0: Microsoft Corporation
C# 1.2	Download DOC	C# Language Specification Version 1.2: Microsoft Corporation
C# 2.0	Download PDF	Standard ECMA-334 4th Edition
C# 3.0	Download DOC	C# Language Specification Version 3.0: Microsoft Corporation
C# 5.0	Download PDF	Standard ECMA-334 5th Edition
C# 6.0	Link	C# Language Specification Version 6 - Unofficial Draft: .NET Foundation
C# 7.0 and later		not currently available

Minimum compiler version needed to support all language features

↩ISO1: Microsoft Visual Studio/Build Tools .Net 2002 or bundled .Net Framework 1.0 compiler ↩ISO2: Microsoft

Visual Studio/Build Tools 2005 or bundled .Net Framework 2.0 compiler ↩CS3: Microsoft Visual Studio/Build Tools 2008 or bundled .Net Framework 3.5 compiler ↩CS4: Microsoft Visual Studio/Build Tools 2010 or bundled .Net Framework 4.0 compiler ↩CS5: Microsoft Visual Studio/Build Tools 2012 or bundled .Net Framework 4.5 compiler ↩CS6: Microsoft Visual Studio/Build Tools 2015 ↩CS7: Microsoft Visual Studio/Build Tools 2017 ↩CS71: Microsoft Visual Studio/Build Tools 2017, version 15.3 ↩CS72: Microsoft Visual Studio/Build Tools 2017, version 15.5 ↩CS73: Microsoft Visual Studio/Build Tools 2017, version 15.7

-lib (C# Compiler Options)

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

The **-lib** option specifies the location of assemblies referenced by means of the -reference (C# Compiler Options) option.

Syntax

```
-lib:dir1[,dir2]
```

Arguments

`dir1`

A directory for the compiler to look in if a referenced assembly is not found in the current working directory (the directory from which you are invoking the compiler) or in the common language runtime's system directory.

`dir2`

One or more additional directories to search in for assembly references. Separate additional directory names with a comma, and without white space between them.

Remarks

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by **-lib**.
4. Directories specified by the LIB environment variable.

Use **-reference** to specify an assembly reference.

-lib is additive; specifying it more than once appends to any prior values.

An alternative to using **-lib** is to copy into the working directory any required assemblies; this will allow you to simply pass the assembly name to **-reference**. You can then delete the assemblies from the working directory. Since the path to the dependent assembly is not specified in the assembly manifest, the application can be started on the target computer and will find and use the assembly in the global assembly cache.

Because the compiler can reference the assembly does not imply the common language runtime will be able to find and load the assembly at runtime. See How the Runtime Locates Assemblies for details on how the runtime searches for referenced assemblies.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box.
2. Click the **References Path** property page.
3. Modify the contents of the list box.

For information on how to set this compiler option programmatically, see ReferencePath.

Example

Compile t2.cs to create an .exe file. The compiler will look in the working directory and in the root directory of the C drive for assembly references.

```
csc -lib:c:\ -reference:t2.dll t2.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-link (C# Compiler Options)

1/23/2019 • 3 minutes to read • Edit Online

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

Syntax

```
-link:fileList
// -or-
-l:fileList
```

Arguments

`fileList`
Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

Remarks

The `-link` option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see Walkthrough: Embedding Types from Managed Assemblies.

Using the `-link` option is especially useful when you are working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The `-link` option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types.

The `-link` option embeds only interfaces, structures, and delegates. Embedding COM classes is not supported.

> **NOTE**
> When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the CoClass causes an error.

To set the `-link` option in Visual Studio, add an assembly reference and set the `Embed Interop Types` property to **true**. The default for the `Embed Interop Types` property is **false**.

If you link to a COM assembly (Assembly A) which itself references another COM assembly (Assembly B), you also have to link to Assembly B if either of the following is true:

- A type from Assembly A inherits from a type or implements an interface from Assembly B.

- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Like the -reference compiler option, the `-link` compiler option uses the Csc.rsp response file, which references frequently used .NET Framework assemblies. Use the -noconfig compiler option if you do not want the compiler to use the Csc.rsp file.

The short form of `-link` is `-l`.

Generics and Embedded Types

The following sections describe the limitations on using generic types in applications that embed interop types.

Generic Interfaces

Generic interfaces that are embedded from an interop assembly cannot be used. This is shown in the following example.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

Types That Have Generic Parameters

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction does not apply to interfaces. For example, consider the Range interface that is defined in the Microsoft.Office.Interop.Excel assembly. If a library embeds interop types from the Microsoft.Office.Interop.Excel assembly and exposes a method that returns a generic type that has a parameter whose type is the Range interface, that method must return a generic interface, as shown in the following code example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1() {
```

```
}

// The following code is valid for calls from a client assembly.
public IList<Range> GetRange2() {
```

```
    }
}
```

In the following example, client code can call the method that returns the IList generic interface without error.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

Example

The following code compiles source file `OfficeApp.cs` and reference assemblies from `COMData1.dll` and `COMData2.dll` to produce `OfficeApp.exe`.

```
csc -link:COMData1.dll,COMData2.dll -out:OfficeApp.exe OfficeApp.cs
```

See also

- C# Compiler Options
- Walkthrough: Embedding Types from Managed Assemblies
- -reference (C# Compiler Options)
- -noconfig (C# Compiler Options)
- Command-line Building With csc.exe
- Interoperability Overview

-linkresource (C# Compiler Options)

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

Creates a link to a .NET Framework resource in the output file. The resource file is not added to the output file. This differs from the -resource option which does embed a resource file in the output file.

Syntax

```
-linkresource:filename[,identifier[,accessibility-modifier]]
```

Arguments

`filename`
The .NET Framework resource file to which you want to link from the assembly.

`identifier` (optional)
The logical name for the resource; the name that is used to load the resource. The default is the name of the file.

`accessibility-modifier` (optional)
The accessibility of the resource: public or private. The default is public.

Remarks

By default, linked resources are public in the assembly when they are created with the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other modifier other than `public` or `private` is allowed.

-linkresource requires one of the -target options other than **-target:module**.

If `filename` is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource` methods in the Assembly class to access the resource at run time.

The file specified in `filename` can be any format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly. The second of the following examples shows how to do this. You can do the same thing in the Assembly Linker. The third of the following examples shows how to do this. For more information, see Al.exe (Assembly Linker) and Working with Assemblies and the Global Assembly Cache.

-linkres is the short form of **-linkresource**.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

Compile `in.cs` and link to resource file `rf.resource`:

```
csc -linkresource:rf.resource in.cs
```

Example

Compile `A.cs` into a DLL, link to a native DLL N.dll, and put the output in the Global Assembly Cache (GAC). In this example, both A.dll and N.dll will reside in the GAC.

```
csc -linkresource:N.dll -t:library A.cs
gacutil -i A.dll
```

Example

This example does the same thing as the previous one, but by using Assembly Linker options.

```
csc -t:module A.cs
al -out:A.dll A.netmodule -link:N.dll
gacutil -i A.dll
```

See also

- C# Compiler Options
- Al.exe (Assembly Linker)
- Working with Assemblies and the Global Assembly Cache
- Managing Project and Solution Properties

-main (C# Compiler Options)

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

This option specifies the class that contains the entry point to the program, if more than one class contains a **Main** method.

Syntax

```
-main:class
```

Arguments

`class`
The type that contains the **Main** method.
The provided class name must be fully qualified; it must include the full namespace containing the class, followed by the class name. For example, when the `Main` method is located inside the `Program` class in the `MyApplication.Core` namespace, the compiler option has to be `-main:MyApplication.Core.Program`.

Remarks

If your compilation includes more than one type with a Main method, you can specify which type contains the **Main** method that you want to use as the entry point into the program.

This option is for use when compiling an .exe file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Modify the **Startup object** property.

   To set this compiler option programmatically, see StartupObject.

Example

Compile `t2.cs` and `t3.cs`, specifying that the **Main** method will be found in `Test2`:

```
csc t2.cs t3.cs -main:Test2
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-moduleassemblyname (C# Compiler Option)

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

Specifies an assembly whose non-public types a .netmodule can access.

Syntax

```
-moduleassemblyname:assembly_name
```

Arguments

`assembly_name`
The name of the assembly whose non-public types the .netmodule can access.

Remarks

-moduleassemblyname should be used when building a .netmodule, and where the following conditions are true:

- The .netmodule needs access to non-public types in an existing assembly.
- You know the name of the assembly into which the .netmodule will be built.
- The existing assembly has granted friend assembly access to the assembly into which the .netmodule will be built.

For more information on building a .netmodule, see -target:module (C# Compiler Options).

For more information on friend assemblies, see Friend Assemblies.

This option is not available from within the development environment; it is only available when compiling from the command line.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

This sample builds an assembly with a private type, and that gives friend assembly access to an assembly called csman_an_assembly.

```
// moduleassemblyname_1.cs
// compile with: -target:library
using System;
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo ("csman_an_assembly")]

class An_Internal_Class
{
    public void Test()
    {
        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

Example

This sample builds a .netmodule that accesses a non-public type in the assembly moduleassemblyname_1.dll. By knowing that this .netmodule will be built into an assembly called csman_an_assembly, we can specify **-moduleassemblyname**, allowing the .netmodule to access non-public types in an assembly that has granted friend assembly access to csman_an_assembly.

```
// moduleassemblyname_2.cs
// compile with: -moduleassemblyname:csman_an_assembly -target:module -reference:moduleassemblyname_1.dll
class B {
    public void Test() {
        An_Internal_Class x = new An_Internal_Class();
        x.Test();
    }
}
```

Example

This code sample builds the assembly csman_an_assembly, referencing the previously-built assembly and .netmodule.

```
// csman_an_assembly.cs
// compile with: -addmodule:moduleassemblyname_2.netmodule -reference:moduleassemblyname_1.dll
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

An_Internal_Class.Test called

See also

- C# Compiler Options
- Managing Project and Solution Properties

-noconfig (C# Compiler Options)

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

The **-noconfig** option tells the compiler not to compile with the csc.rsp file, which is located in and loaded from the same directory as the csc.exe file.

Syntax

```
-noconfig
```

Remarks

The csc.rsp file references all the assemblies shipped with the .NET Framework. The actual references that the Visual Studio .NET development environment includes depend on the project type.

You can modify the csc.rsp file and specify additional compiler options that should be included in every compilation from the command line with csc.exe (except the **-noconfig** option).

The compiler processes the options passed to the **csc** command last. Therefore, any option on the command line overrides the setting of the same option in the csc.rsp file.

If you do not want the compiler to look for and use the settings in the csc.rsp file, specify **-noconfig**.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-nologo (C# Compiler Options)

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

The **-nologo** option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

Syntax

```
-nologo
```

Remarks

This option is not available from within the development environment; it is only available when compiling from the command line.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-nostdlib (C# Compiler Options)

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

-nostdlib prevents the import of mscorlib.dll, which defines the entire System namespace.

Syntax

```
-nostdlib[+ | -]
```

Remarks

Use this option if you want to define or create your own System namespace and objects.

If you do not specify **-nostdlib**, mscorlib.dll is imported into your program (same as specifying **-nostdlib-**). Specifying **-nostdlib** is the same as specifying **-nostdlib+**.

To set this compiler option in Visual Studio

> **NOTE**
> The following instructions apply to Visual Studio 2015 (and earlier versions) only. The **Do not reference mscorlib.dll** build property doesn't exist in Visual Studio 2017.

1. Open the **Properties** page for the project.
2. Click the **Build** properties page.
3. Click the **Advanced** button.
4. Modify the **Do not reference mscorlib.dll** property.

To set this compiler option programmatically

For information on how to set this compiler option programmatically, see NoStdLib.

See also

- C# Compiler Options

-nowarn (C# Compiler Options)

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

The **-nowarn** option lets you suppress the compiler from displaying one or more warnings. Separate multiple warning numbers with a comma.

Syntax

```
-nowarn:number1[,number2,...]
```

Arguments

`number1`, `number2`
Warning number(s) that you want the compiler to suppress.

Remarks

You should only specify the numeric part of the warning identifier. For example, if you want to suppress CS0028, you could specify `-nowarn:28`.

The compiler will silently ignore warning numbers passed to `-nowarn` that were valid in previous releases, but that have been removed from the compiler. For example, CS0679 was valid in the compiler in Visual Studio .NET 2002 but was subsequently removed.

The following warnings cannot be suppressed by the `-nowarn` option:

- Compiler Warning (level 1) CS2002
- Compiler Warning (level 1) CS2023
- Compiler Warning (level 1) CS2029

To set this compiler option in the Visual Studio development environment

1. Open the **Properties** page for the project. For details, see Build Page, Project Designer (C#).
2. Click the **Build** property page.
3. Modify the **Suppress Warnings** property.

For information about how to set this compiler option programmatically, see DelaySign.

See also

- C# Compiler Options
- Managing Project and Solution Properties
- C# Compiler Errors

-nowin32manifest (C# Compiler Options)

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

Use the **-nowin32manifest** option to instruct the compiler not to embed any application manifest into the executable file.

Syntax

```
-nowin32manifest
```

Remarks

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step.

In Visual Studio, set this option in the **Application Property** page by selecting the **Create Application Without a Manifest** option in the **Manifest** drop down list. For more information, see Application Page, Project Designer (C#).

For more information about manifest creation, see -win32manifest (C# Compiler Options).

See also

- C# Compiler Options
- Managing Project and Solution Properties

-optimize (C# Compiler Options)

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

The **-optimize** option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient.

Syntax

```
-optimize[+ | -]
```

Remarks

-optimize also tells the common language runtime to optimize code at runtime.

By default, optimizations are disabled. Specify **-optimize+** to enable optimizations.

When building a module to be used by an assembly, use the same **-optimize** settings as those of the assembly.

-o is the short form of **-optimize**.

It is possible to combine the **-optimize** and -debug options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Modify the **Optimize Code** property.

For information on how to set this compiler option programmatically, see Optimize.

Example

Compile `t2.cs` and enable compiler optimizations:

```
csc t2.cs -optimize
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-out (C# Compiler Options)

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

The **-out** option specifies the name of the output file.

Syntax

```
-out:filename
```

Arguments

`filename`
The name of the output file created by the compiler.

Remarks

On the command line, it is possible to specify multiple output files for your compilation. The compiler expects to find one or more source code files following the **-out** option. Then, all source code files will be compiled into the output file specified by that **-out** option.

Specify the full name and extension of the file you want to create.

If you do not specify the name of the output file:

- An .exe will take its name from the source code file that contains the **Main** method.
- A .dll or .netmodule will take its name from the first source code file.

A source code file used to compile one output file cannot be used in the same compilation for the compilation of another output file.

When producing multiple output files in a command-line compilation, keep in mind that only one of the output files can be an assembly and that only the first output file specified (implicitly or explicitly with **-out**) can be the assembly.

Any modules produced as part of a compilation become files associated with any assembly also produced in the compilation. Use ildasm.exe to view the assembly manifest to see the associated files.

The -out compiler option is required in order for an exe to be the target of a friend assembly. For more information see Friend Assemblies.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Modify the **Assembly name** property.

   To set this compiler option programmatically: the OutputFileName is a read-only property, which is determined by a combination of the project type (exe, library, and so forth) and the assembly name. Modifying one or both of these properties will be necessary to set the output file name.

Example

Compile `t.cs` and create output file `t.exe`, as well as build `t2.cs` and create module output file `mymodule.netmodule`:

```
csc t.cs -out:mymodule.netmodule -target:module t2.cs
```

See also

- C# Compiler Options
- Friend Assemblies
- Managing Project and Solution Properties

-pathmap (C# Compiler Options)

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

The **-pathmap** compiler option specifies how to map physical paths to source path names output by the compiler.

Syntax

```
-pathmap:path1=sourcePath1,path2=sourcePath2
```

Arguments

`path1` The full path to the source files in the current environment

`sourcePath1` The source path substituted for `path1` in any output files.

To specify multiple mapped source paths, separate each with a comma.

Remarks

The compiler writes the source path into its output for the following reasons:

1. The source path is substituted for an argument when the CallerFilePathAttribute is applied to an optional parameter.
2. The source path is embedded in a PDB file.
3. The path of the PDB file is embedded into a PE (portable executable) file.

This option maps each physical path on the machine where the compiler runs to a corresponding path that should be written in the output files.

Example

Compile `t.cs` in the directory **C:\work\tests** and map that directory to **\publish** in the output:

```
csc -pathmap:C:\work\tests=\publish t.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-pdb (C# Compiler Options)

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

The **-pdb** compiler option specifies the name and location of the debug symbols file.

Syntax

```
-pdb:filename
```

Arguments

`filename`
The name and location of the debug symbols file.

Remarks

When you specify -debug (C# Compiler Options), the compiler will create a .pdb file in the same directory where the compiler will create the output file (.exe or .dll) with a file name that is the same as the name of the output file.

-pdb allows you to specify a non-default file name and location for the .pdb file.

This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

Example

Compile `t.cs` and create a .pdb file called tt.pdb:

```
csc -debug -pdb:tt t.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-platform (C# Compiler Options)

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

Specifies which version of the Common Language Runtime (CLR) can run the assembly.

Syntax

```
-platform:string
```

Parameters

`string`

anycpu (default), anycpu32bitpreferred, ARM, x64, x86, or Itanium.

Remarks

- **anycpu** (default) compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.
- **anycpu32bitpreferred** compiles your assembly to run on any platform. Your application runs in 32-bit mode on systems that support both 64-bit and 32-bit applications. You can specify this option only for projects that target the .NET Framework 4.5.
- **ARM** compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.
- **x64** compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
- **x86** compiles your assembly to be run by the 32-bit, x86-compatible CLR.
- **Itanium** compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.

On a 64-bit Windows operating system:

- Assemblies compiled with **-platform:x86** execute on the 32-bit CLR running under WOW64.
- A DLL compiled with the **-platform:anycpu** executes on the same CLR as the process into which it is loaded.
- Executables that are compiled with the **-platform:anycpu** execute on the 64-bit CLR.
- Executables compiled with **-platform:anycpu32bitpreferred** execute on the 32-bit CLR.

The **anycpu32bitpreferred** setting is valid only for executable (.EXE) files, and it requires the .NET Framework 4.5.

For more information about developing an application to run on a Windows 64-bit operating system, see 64-bit Applications.

To set this compiler option in the Visual Studio development environment

1. Open the **Properties** page for the project.
2. Click the **Build** property page.
3. Modify the **Platform target** property and, for projects that target the .NET Framework 4.5, select or clear

the **Prefer 32-bit** check box.

Note **-platform** is not available in the development environment in Visual C# Express.

For information on how to set this compiler option programmatically, see PlatformTarget.

Example

The following example shows how to use the **-platform** option to specify that the application should be run by the 64-bit CLR on a 64-bit Windows operating system.

```
csc -platform:anycpu filename.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-preferreduilang (C# Compiler Options)

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

By using the `-preferreduilang` compiler option, you can specify the language in which the C# compiler displays output, such as error messages.

Syntax

```
-preferreduilang: language
```

Arguments

`language`
The language name of the language to use for compiler output.

Remarks

You can use the `-preferreduilang` compiler option to specify the language that you want the C# compiler to use for error messages and other command-line output. If the language pack for the language is not installed, the language setting of the operating system is used instead, and no error is reported.

```
csc.exe -preferreduilang:ja-JP
```

Requirements

See also

- C# Compiler Options

-publicsign (C# Compiler Options)

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

This option causes the compiler to apply a public key but does not actually sign the assembly. The **-publicsign** option also sets a bit in the assembly that tells the runtime that the file is actually signed.

Syntax

```
-publicsign
```

Arguments

None.

Remarks

The **-publicsign** option requires the use of the -keyfile or -keycontainer. The **keyfile** or **keycontainer** options specify the public key.

The **-publicsign** and **-delaysign** options are mutually exclusive.

Sometimes called "fake sign" or "OSS sign", public signing includes the public key in an output assembly and sets the "signed" flag, but doesn't actually sign the assembly with a private key. This is useful for open source projects where people want to build assemblies which are compatible with the released "fully signed" assemblies, but don't have access to the private key used to sign the assemblies. Since almost no consumers actually need to check if the assembly is fully signed, these publicly built assemblies are useable in almost every scenario where the fully signed one would be used.

To set this compiler option in the Visual Studio development environment

1. Open the **Properties** page for the project.
2. Modify the **Delay sign only** property.

See also

- C# Compiler -delaysign option
- C# Compiler -keyfile option
- C# Compiler -keycontainer option
- C# Compiler Options
- Managing Project and Solution Properties

-recurse (C# Compiler Options)

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

The -recurse option enables you to compile source code files in all child directories of either the specified directory (dir) or of the project directory.

Syntax

```
-recurse:[dir\]file
```

Arguments

`dir` (optional)
The directory in which you want the search to begin. If this is not specified, the search begins in the project directory.

`file`
The file(s) to search for. Wildcard characters are allowed.

Remarks

The **-recurse** option lets you compile source code files in all child directories of either the specified directory (`dir`) or of the project directory.

You can use wildcards in a file name to compile all matching files in the project directory without using **-recurse**.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

Compiles all C# files in the current directory:

```
csc *.cs
```

Compiles all of the C# files in the dir1\dir2 directory and any directories below it and generates dir2.dll:

```
csc -target:library -out:dir2.dll -recurse:dir1\dir2\*.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-reference (C# Compiler Options)

1/23/2019 • 3 minutes to read • Edit Online

The **-reference** option causes the compiler to import public type information in the specified file into the current project, thus enabling you to reference metadata from the specified assembly files.

Syntax

```
-reference:[alias=]filename
-reference:filename
```

Arguments

`filename`
The name of a file that contains an assembly manifest. To import more than one file, include a separate **-reference** option for each file.

`alias`
A valid C# identifier that will represent a root namespace that will contain all namespaces in the assembly.

Remarks

To import from more than one file, include a **-reference** option for each file.

The files you import must contain a manifest; the output file must have been compiled with one of the -target options other than -target:module.

-r is the short form of **-reference**.

Use -addmodule to import metadata from an output file that does not contain an assembly manifest.

If you reference an assembly (Assembly A) that references another assembly (Assembly B), you will need to reference Assembly B if:

- A type you use from Assembly A inherits from a type or implements an interface from Assembly B.
- You invoke a field, property, event, or method that has a return type or parameter type from Assembly B.

Use -lib to specify the directory in which one or more of your assembly references is located. The **-lib** topic also discusses the directories in which the compiler searches for assemblies.

In order for the compiler to recognize a type in an assembly, and not in a module, it needs to be forced to resolve the type, which you can do by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler: for example, if you inherit from a type in an assembly, the type name will then be recognized by the compiler.

Sometimes it is necessary to reference two different versions of the same component from within one assembly. To do this, use the alias suboption on the **-reference** switch for each file to distinguish between the two files. This alias will be used as a qualifier for the component name, and will resolve to the component in one of the files.

The csc response (.rsp) file, which references commonly used .NET Framework assemblies, is used by default. Use -noconfig if you do not want the compiler to use csc.rsp.

NOTE

In Visual Studio, use the **Add Reference** dialog box. For more information, see How to: Add or Remove References By Using the Reference Manager. To ensure equivalent behavior between adding references by using `-reference` and adding references by using the **Add Reference** dialog box, set the **Embed Interop Types** property to **False** for the assembly that you're adding. **True** is the default value for the property.

Example

This example shows how to use the extern alias feature.

You compile the source file and import metadata from `grid.dll` and `grid20.dll`, which have been compiled previously. The two DLLs contain separate versions of the same component, and you use two **-reference** with alias options to compile the source file. The options look like this:

```
-reference:GridV1=grid.dll -reference:GridV2=grid20.dll
```

This sets up the external aliases `GridV1` and `GridV2`, which you use in your program by means of an `extern` statement:

```
extern alias GridV1;
extern alias GridV2;
// Using statements go here.
```

Once this is done, you can refer to the grid control from `grid.dll` by prefixing the control name with `GridV1`, like this:

```
GridV1::Grid
```

In addition, you can refer to the grid control from `grid20.dll` by prefixing the control name with `GridV2` like this:

```
GridV2::Grid
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-refout (C# Compiler Options)

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

The **-refout** option specifies a file path where the reference assembly should be output. This translates to `metadataPeStream` in the Emit API.

Syntax

```
-refout:filepath
```

Arguments

`filepath` The filepath for the reference assembly. It should generally match that of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" sub-folder relative to the primary assembly.

Remarks

Metadata-only assemblies have their method bodies replaced with a single `throw null` body, but include all members except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level `ReferenceAssembly` attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in reflection-only mode). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only, otherwise they will receive a typeload error from the runtime.

Reference assemblies further remove metadata (private members) from metadata-only assemblies:

- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations. For instance, the reference assembly for `class C { private void M() { dynamic d = 1; ... } }` does not reference any types required for `dynamic`.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no `InternalsVisibleTo` attributes, do the same for internal function-members.
- But all types (including private or nested types) are kept in reference assemblies. All attributes are kept (even internal ones).
- All virtual methods are kept. Explicit interface implementations are kept. Explicitly implemented properties and events are kept, as their accessors are virtual (and are therefore kept).
- All fields of a struct are kept. (This is a candidate for post-C#-7.1 refinement)

The `-refout` and `-refonly` options are mutually exclusive.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-refonly (C# Compiler Options)

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

The **-refonly** option indicates that a reference assembly should be output instead of an implementation assembly, as the primary output. The `-refonly` parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

Syntax

```
-refonly
```

Remarks

Metadata-only assemblies have their method bodies replaced with a single `throw null` body, but include all members except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level `ReferenceAssembly` attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in reflection-only mode). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only, otherwise they will receive a typeload error from the runtime.

Reference assemblies further remove metadata (private members) from metadata-only assemblies:

- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations. For instance, the reference assembly for `class C { private void M() { dynamic d = 1; ... } }` does not reference any types required for `dynamic`.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no `InternalsVisibleTo` attributes, do the same for internal function-members.
- But all types (including private or nested types) are kept in reference assemblies. All attributes are kept (even internal ones).
- All virtual methods are kept. Explicit interface implementations are kept. Explicitly implemented properties and events are kept, as their accessors are virtual (and are therefore kept).
- All fields of a struct are kept. (This is a candidate for post-C#-7.1 refinement)

The `-refonly` and `-refout` options are mutually exclusive.

See also

- C# Compiler Options
- Managing Project and Solution Properties

-resource (C# Compiler Options)

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

Embeds the specified resource into the output file.

Syntax

```
-resource:filename[,identifier[,accessibility-modifier]]
```

Arguments

`filename`
The .NET Framework resource file that you want to embed in the output file.

`identifier` (optional)
The logical name for the resource; the name that is used to load the resource. The default is the name of the file name.

`accessibility-modifier` (optional)
The accessibility of the resource: public or private. The default is public.

Remarks

Use -linkresource to link a resource to an assembly and not add the resource file to the output file.

By default, resources are public in the assembly when they are created by using the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other accessibility other than `public` or `private` is allowed.

If `filename` is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource` methods in the Assembly class to access the resource at run time.

-res is the short form of **-resource**.

The order of the resources in the output file is determined from the order specified on the command line.

To set this compiler option in the Visual Studio development environment

1. Add a resource file to your project.
2. Select the file that you want to embed in **Solution Explorer**.
3. Select **Build Action** for the file in the **Properties** window.
4. Set **Build Action** to **Embedded Resource**.

For information about how to set this compiler option programmatically, see BuildAction.

Example

Compile `in.cs` and attach resource file `rf.resource`:

```
csc -resource:rf.resource in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-subsystemversion (C# Compiler Options)

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

Specifies the minimum version of the subsystem on which the generated executable file can run, thereby determining the versions of Windows on which the executable file can run. Most commonly, this option ensures that the executable file can leverage particular security features that aren't available with older versions of Windows.

> **NOTE**
>
> To specify the subsystem itself, use the -target compiler option.

Syntax

```
-subsystemversion:major.minor
```

Parameters

`major.minor`

The minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7 if you set the value of this option to 6.01, as the table later in this topic describes. You must specify the values for `major` and `minor` as integers.

Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

Remarks

The following table lists common subsystem versions of Windows.

WINDOWS VERSION	SUBSYSTEM VERSION
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Default values

The default value of the **-subsystemversion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
  - -target:appcontainerexe
  - -target:winmdobj
  - -platform:arm
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions is true.

Setting this option

To set the **-subsystemversion** compiler option in Visual Studio, you must open the .csproj file and specify a value for the `SubsystemVersion` property in the MSBuild XML. You can't set this option in the Visual Studio IDE. For more information, see "Default values" earlier in this topic or Common MSBuild Project Properties.

See also

- C# Compiler Options

-target (C# Compiler Options)

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

The **-target** compiler option can be specified in one of four forms:

-target:appcontainerexe
To create an .exe file for Windows 8.x Store apps.

-target:exe
To create an .exe file.

-target:library
To create a code library.

-target:module
To create a module.

-target:winexe
To create a Windows program.

-target:winmdobj
To create an intermediate .winmdobj file.

Unless you specify **-target:module**, **-target** causes a .NET Framework assembly manifest to be placed in an output file. For more information, see Assemblies in the Common Language Runtime and Common Attributes.

The assembly manifest is placed in the first .exe output file in the compilation or in the first DLL, if there is no .exe output file. For example, in the following command line, the manifest will be placed in `1.exe`:

```
csc -out:1.exe t1.cs -out:2.netmodule t2.cs
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. All output files except those created with **-target:module** can contain an assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line. No matter what the first output file is (**-target:exe**, **-target:winexe**, **-target:library** or **-target:module**), any other output files produced in the same compilation must be modules (**-target:module**).

If you create an assembly, you can indicate that all or part of your code is CLS compliant with the CLSCompliantAttribute attribute.

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

[System.CLSCompliant(false)]   // specify compliance for an element
public class TestClass
{
    public static void Main() {}
}
```

For more information about setting this compiler option programmatically, see OutputType.

See also

- C# Compiler Options
- Managing Project and Solution Properties
- -subsystemversion (C# Compiler Options)

-target:appcontainerexe (C# Compiler Options)

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

If you use the **-target:appcontainerexe** compiler option, the compiler creates a Windows executable (.exe) file that must be run in an app container. This option is equivalent to -target:winexe but is designed for Windows 8.x Store apps.

Syntax

```
-target:appcontainerexe
```

Remarks

To require the app to run in an app container, this option sets a bit in the Portable Executable (PE) file. When that bit is set, an error occurs if the CreateProcess method tries to launch the executable file outside an app container.

Unless you use the -out option, the output file name takes the name of the input file that contains the Main method.

When you specify this option at a command prompt, all files until the next **-out** or **-target** option are used to create the executable file.

To set this compiler option in the IDE

1. In **Solution Explorer**, open the shortcut menu for your project, and then choose **Properties**.
2. On the **Application** tab, in the **Output type** list, choose **Windows Store App**.

   This option is available only for Windows 8.x Store app templates.

For information about how to set this compiler option programmatically, see OutputType.

Example

The following command compiles `filename.cs` into a Windows executable file that can be run only in an app container.

```
csc -target:appcontainerexe filename.cs
```

See also

- -target (C# Compiler Options)
- -target:winexe (C# Compiler Options)
- C# Compiler Options

-target:exe (C# Compiler Options)

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

The **-target:exe** option causes the compiler to create an executable (EXE), console application.

Syntax

```
-target:exe
```

Remarks

The **-target:exe** option is in effect by default. The executable file will be created with the .exe extension.

Use -target:winexe to create a Windows program executable.

Unless otherwise specified with the -out option, the output file name takes the name of the input file that contains the Main method.

When specified at the command line, all files up to the next **-out** or **-target:module** option are used to create the .exe file

One and only one **Main** method is required in the source code files that are compiled into an .exe file. The -main compiler option lets you specify which class contains the **Main** method, in cases where your code has more than one class with a **Main** method.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Modify the **Output type** property.

For information on how to set this compiler option programmatically, see OutputType.

Example

Each of the following command lines will compile `in.cs`, creating `in.exe`:

```
csc -target:exe in.cs
csc in.cs
```

See also

- -target (C# Compiler Options)
- C# Compiler Options

-target:library (C# Compiler Options)

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

The **-target:library** option causes the compiler to create a dynamic-link library (DLL) rather than an executable file (EXE).

Syntax

```
-target:library
```

Remarks

The DLL will be created with the .dll extension.

Unless otherwise specified with the -out option, the output file name takes the name of the first input file.

When specified at the command line, all files up to the next **-out** or **-target:module** option are used to create the .dll file.

When building a .dll file, a Main method is not required.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Modify the **Output type** property.

For information on how to set this compiler option programmatically, see OutputType.

Example

Compile `in.cs`, creating `in.dll`:

```
csc -target:library in.cs
```

See also

- -target (C# Compiler Options)
- C# Compiler Options

-target:module (C# Compiler Options)

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

This option causes the compiler to not generate an assembly manifest.

Syntax

```
-target:module
```

Remarks

By default, the output file created by compiling with this option will have an extension of .netmodule.

A file that does not have an assembly manifest cannot be loaded by the .NET Framework common language runtime. However, such a file can be incorporated into the assembly manifest of an assembly by means of -addmodule.

If more than one module is created in a single compilation, internal types in one module will be available to other modules in the compilation. When code in one module references `internal` types in another module, then both modules must be incorporated into an assembly manifest, by means of **-addmodule**.

Creating a module is not supported in the Visual Studio development environment.

For information on how to set this compiler option programmatically, see OutputType.

Example

Compile `in.cs`, creating `in.netmodule`:

```
csc -target:module in.cs
```

See also

- -target (C# Compiler Options)
- C# Compiler Options

-target:winexe (C# Compiler Options)

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

The **-target:winexe** option causes the compiler to create an executable (EXE), Windows program.

Syntax

```
-target:winexe
```

Remarks

The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from either the .NET Framework library or with the Win32 APIs.

Use -target:exe to create a console application.

Unless otherwise specified with the -out option, the output file name takes the name of the input file that contains the Main method.

When specified at the command line, all files until the next **-out** or -target option are used to create the Windows program.

One and only one **Main** method is required in the source code files that are compiled into an .exe file. The -main option lets you specify which class contains the **Main** method, in cases where your code has more than one class with a **Main** method.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Modify the **Output type** property.

For information on how to set this compiler option programmatically, see OutputType.

Example

Compile `in.cs` into a Windows program:

```
csc -target:winexe in.cs
```

See also

- -target (C# Compiler Options)
- C# Compiler Options

-target:winmdobj (C# Compiler Options)

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

If you use the **-target:winmdobj** compiler option, the compiler creates an intermediate .winmdobj file that you can convert to a Windows Runtime binary (.winmd) file. The .winmd file can then be consumed by JavaScript and C++ programs, in addition to managed language programs.

Syntax

```
-target:winmdobj
```

Remarks

The **winmdobj** setting signals to the compiler that an intermediate module is required. In response, Visual Studio compiles the C# class library as a .winmdobj file. The .winmdobj file can then be fed through the WinMDExp export tool to produce a Windows metadata (.winmd) file. The .winmd file contains both the code from the original library and the WinMD metadata that is used by JavaScript or C++ and by the Windows Runtime.

The output of a file that's compiled by using the **-target:winmdobj** compiler option is designed to be used only as input for the WimMDExp export tool; the .winmdobj file itself isn't referenced directly.

Unless you use the -out option, the output file name takes the name of the first input file. A Main method isn't required.

If you specify the -target:winmdobj option at a command prompt, all files until the next **-out** or -target:module option are used to create the Windows program.

To set this compiler option in the Visual Studio IDE for a Windows Store app

1. In **Solution Explorer**, open the shortcut menu for your project, and then choose **Properties**.
2. Choose the **Application** tab.
3. In the **Output type** list, choose **WinMD File**.

   The **WinMD File** option is available only for Windows 8.x Store app templates.

For information about how to set this compiler option programmatically, see OutputType.

Example

The following command compiles `filename.cs` into an intermediate .winmdobj file.

```
csc -target:winmdobj filename.cs
```

See also

- -target (C# Compiler Options)
- C# Compiler Options

-unsafe (C# Compiler Options)

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

The **-unsafe** compiler option allows code that uses the unsafe keyword to compile.

Syntax

```
-unsafe
```

Remarks

For more information about unsafe code, see Unsafe Code and Pointers.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Select the **Allow Unsafe Code** check box.

To add this option in a csproj file

Open the .csproj file for a project, and add the following elements:

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

For information about how to set this compiler option programmatically, see AllowUnsafeBlocks.

Example

Compile `in.cs` for unsafe mode:

```
csc -unsafe in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-utf8output (C# Compiler Options)

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

The **-utf8output** option displays compiler output using UTF-8 encoding.

Syntax

```
-utf8output
```

Remarks

In some international configurations, compiler output cannot correctly be displayed in the console. In these configurations, use **-utf8output** and redirect compiler output to a file.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

See also

- C# Compiler Options

-warn (C# Compiler Options)

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

The **-warn** option specifies the warning level for the compiler to display.

Syntax

```
-warn:option
```

Arguments

`option`

The warning level you want displayed for the compilation: Lower numbers show only high severity warnings; higher numbers show more warnings. Valid values are 0-4:

WARNING LEVEL	MEANING
0	Turns off emission of all warning messages.
1	Displays severe warning messages.
2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to `true` or `false`.
4 (the default)	Displays all level 3 warnings plus informational warnings.

Remarks

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see C# Compiler Errors.

Use -warnaserror to treat all warnings as errors. Use -nowarn to disable certain warnings.

-w is the short form of **-warn**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Modify the **Warning Level** property.

For information on how to set this compiler option programmatically, see WarningLevel.

Example

Compile `in.cs` and have the compiler only display level 1 warnings:

```
csc -warn:1 in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-warnaserror (C# Compiler Options)

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

The **-warnaserror+** option treats all warnings as errors

Syntax

```
-warnaserror[+ | -][:warning-list]
```

Remarks

Any messages that would ordinarily be reported as warnings are instead reported as errors, and the build process is halted (no output files are built).

By default, **-warnaserror-** is in effect, which causes warnings to not prevent the generation of an output file. **-warnaserror**, which is the same as **-warnaserror+**, causes warnings to be treated as errors.

Optionally, if you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors.

Use -warn to specify the level of warnings that you want the compiler to display. Use -nowarn to disable certain warnings.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Build** property page.
3. Modify the **Treat Warnings As Errors** property.

To set this compiler option programmatically, see TreatWarningsAsErrors.

Example

Compile `in.cs` and have the compiler display no warnings:

```
csc -warnaserror in.cs
csc -warnaserror:642,649,652 in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-win32icon (C# Compiler Options)

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

The **-win32icon** option inserts an .ico file in the output file, which gives the output file the desired appearance in the File Explorer.

Syntax

```
-win32icon:filename
```

Arguments

`filename`
The .ico file that you want to add to your output file.

Remarks

An .ico file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file.

See -linkresource (to reference) or -resource (to attach) a .NET Framework resource file. See -win32res to import a .res file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** pages.
2. Click the **Application** property page.
3. Modify the **Application icon** property.

For information on how to set this compiler option programmatically, see ApplicationIcon.

Example

Compile `in.cs` and attach an .ico file `rf.ico` to produce `in.exe`:

```
csc -win32icon:rf.ico in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties

-win32manifest (C# Compiler Options)

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

Use the **-win32manifest** option to specify a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

Syntax

```
-win32manifest: filename
```

Arguments

`filename`

The name and location of the custom manifest file.

Remarks

By default, the Visual C# compiler embeds an application manifest that specifies a requested execution level of "asInvoker." It creates the manifest in the same folder in which the executable is built, typically the bin\Debug or bin\Release folder when you use Visual Studio. If you want to supply a custom manifest, for example to specify a requested execution level of "highestAvailable" or "requireAdministrator," use this option to specify the name of the file.

> **NOTE**
> This option and the -win32res (C# Compiler Options) option are mutually exclusive. If you try to use both options in the same command line you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file/registry virtualization under the User Account Control feature in Windows. For more information, see User Account Control.

Your application will be subject to virtualization if either of these conditions is true:

- You use the **-nowin32manifest** option and you do not provide a manifest in a later build step or as part of a Windows Resource (.res) file by using the **-win32res** option.
- You provide a custom manifest that does not specify a requested execution level.

Visual Studio creates a default .manifest file and stores it in the debug and release directories alongside the executable file. You can add a custom manifest by creating one in any text editor and then adding the file to the project. Alternatively, you can right-click the **Project** icon in **Solution Explorer**, click **Add New Item**, and then click **Application Manifest File**. After you have added your new or existing manifest file, it will appear in the **Manifest** drop down list. For more information, see Application Page, Project Designer (C#).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the -nowin32manifest (C# Compiler Options) option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista. This will prevent the compiler from creating and embedding a default manifest in the portable executable (PE) file.

Example

The following example shows the default manifest that the Visual C# compiler inserts into a PE.

> **NOTE**
>
> The compiler inserts a standard application name " MyApplication.app " into the xml. This is a workaround to enable applications to run on Windows Server 2003 Service Pack 3.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

See also

- C# Compiler Options
- -nowin32manifest (C# Compiler Options)
- Managing Project and Solution Properties

-win32res (C# Compiler Options)

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

The **-win32res** option inserts a Win32 resource in the output file.

Syntax

```
-win32res:filename
```

Arguments

`filename`
The resource file that you want to add to your output file.

Remarks

A Win32 resource file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; a .res file is created from the .rc file.

A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the File Explorer. If you do not specify **-win32res**, the compiler will generate version information based on the assembly version.

See -linkresource (to reference) or -resource (to attach) a .NET Framework resource file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Properties** page.
2. Click the **Application** property page.
3. Click on the **Resource File** button and choose a file by using the combo box.

Example

Compile `in.cs` and attach a Win32 resource file `rf.res` to produce `in.exe`:

```
csc -win32res:rf.res in.cs
```

See also

- C# Compiler Options
- Managing Project and Solution Properties