

#if, #elif, #else, and #endif directives (C/C++)

08/29/2019 • 5 minutes to read •  +1

In this article

[Grammar](#)

[Remarks](#)

[Preprocessor operators](#)

[See also](#)

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is kept in the translation unit.

Grammar

conditional :

if-part elif-parts_{opt} else-part_{opt} endif-line

if-part :

if-line text

if-line :

#if *constant-expression*

#ifdef *identifier*

#ifndef *identifier*

elif-parts :

elif-line text

elif-parts elif-line text

elif-line :

#elif *constant-expression*

else-part :

else-line text

else-line :

#else

endif-line :

#endif

Remarks

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the *text* portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional-compilation directives, such as **#if** and **#ifdef**, must match a closing **#endif** directive before the end of file. Otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is done within the part of the line that follows an **#elif** command, so a macro call can be used in the *constant-expression*.

The preprocessor selects one of the given occurrences of *text* for further processing. A block specified in *text* can be any sequence of text. It can occupy more than one line. Usually *text* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text* and passes it to the compiler. If *text* contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single *text* item by evaluating the constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with **#**) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. When there's no **#else** clause, and all instances of *constant-expression* in the **#if** block are false, no text block is selected.

The *constant-expression* is an integer constant expression with these additional restrictions:

- Expressions must have integral type and can include only integer constants, character constants, and the **defined** operator.
- The expression can't use `sizeof` or a type-cast operator.
- The target environment may be unable to represent all ranges of integers.
- The translation represents type **int** the same way as type **long**, and **unsigned int** the same way as **unsigned long**.
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, use an app built for that environment to check the values of the *LIMITS.H* macros.
- The expression must not query the environment, and must remain insulated from implementation details on the target computer.

Preprocessor operators

defined

The preprocessor operator **defined** can be used in special constant expressions, as shown by the following syntax:

defined(*identifier*)

defined *identifier*


This constant expression is considered true (nonzero) if the *identifier* is currently defined. Otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** operator can be used in an **#if** and an **#elif** directive, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls:

C	 Copy
<pre>#if defined(CREDIT) credit(); #elif defined(DEBIT) debit(); #else perror(); #endif</pre>	


The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `prnterror` is compiled. Both `CREDIT` and `credit` are distinct identifiers in C and C++ because their cases are different.

The conditional compilation statements in the following example assume a previously defined symbolic constant named `DLEVEL`.

C	 Copy
<pre>#if DLEVEL > 5 #define SIGNAL 1 #if STACKUSE == 1 #define STACK 200 #else #define STACK 100 #endif #else #define SIGNAL 0 #if STACKUSE == 1 #define STACK 100 #else #define STACK 50 #endif #endif #if DLEVEL == 0 #define STACK 0 #elif DLEVEL == 1 #define STACK 100 #elif DLEVEL > 5 display(debugptr); #else #define STACK 200 #endif</pre>	


The first **#if** block shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the statements after **#else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

C	 Copy
<pre>#elif DLEVEL > 5 display(debugptr);</pre>	

is compiled, and `STACK` isn't defined.


A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, constructs like this one can be used to prevent multiple definitions:

C++	 Copy
<pre>/* EXAMPLE.H - Example header file */ #if !defined(EXAMPLE_H) #define EXAMPLE_H class Example { //... }; #endif // !defined(EXAMPLE_H)</pre>	

The preceding code checks to see if the symbolic constant `EXAMPLE_H` is defined. If so, the file has already been included and doesn't need reprocessing. If not, the constant `EXAMPLE_H` is defined to mark `EXAMPLE.H` as already processed.

`__has_include`

Visual Studio 2017 version 15.3 and later: Determines whether a library header is available for inclusion:

C++	 Copy
<pre>#ifdef __has_include # if __has_include(<filesystem>) # include <filesystem> # define have_filesystem 1 # elif __has_include(<experimental/filesystem>) # include <experimental/filesystem> # define have_filesystem 1 # define experimental_filesystem # else # define have_filesystem 0 # endif #endif</pre>	

See also

[Preprocessor directives](#)

Is this page helpful?

 Yes  No
