The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.1 Conditional inclusion

conditional inclusion

Constraints

conditional inclusion constant expression

The expression that controls conditional inclusion shall be an integer constant expression except that: it shall not contain a cast; 1869

Commentary

An expression occurring in this context has to be evaluated during translation, hence the need for it to be a constant. The additional constraints are designed to allow the preprocessor to operate independently of the subsequent phases of translation.

Coding Guidelines

source file inclusion

Developers do not commonly refer to this construct using the term *conditional inclusion* (which is suggestive of source file inclusion, i.e., the **#include** directive). Some of the terms that are commonly used by developers include *hash if* and *conditionally compiled*. There does not appear to be a worthwhile benefit in attempting to change existing, developer, use of terminology. However, coding guideline documents still need to make it clear what any terms they use might use, refer to.

it shall not contain a cast; 1870

Commentary

footnote 141 1874

#if operand type uintmax_* 1880

This constraint is redundant for the reasons pointed out in footnote 141 (there is no constraint given here against **sizeof** appearing in this context). While it is possible to write a cast that will evaluate to a constant expression under the rules applied to conditional inclusion expressions (e.g., `(int * const)+1` will evaluate to `1`), in the majority of cases any cast occurring in this context will result in a syntax violation. The preprocessor is not required to have any knowledge of the representation used for any scalar types by subsequent phases of translation. The type of the operands of the constant expression are all defined to have the same rank.

identifiers (including those lexically identical to keywords) are interpreted as described below;[141] 1871

Commentary

#if macros expanded 1876

#if identifier replaced by 0 1878

Identifiers are either subject to macro expansion, treated as an operator (see next C sentence), or replaced by 0.

#if defined

and it may contain unary operator expressions of the form 1872

```
defined identifier
```

or

```
defined ( identifier )
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

Commentary

Semantics in a Constraints clause.

Rationale

The operator **defined** was added to C89 to make possible writing boolean combinations of defined flags with one another and with other inclusion conditions.

The **defined** operator allows more than one test to be made in a single expression. Without this operator it would be necessary to use nested **#ifdef** or **#ifndef** directives, for instance:

```
#ifdef ...
   #ifdef ...
      #ifdef ...
```

Coding Guidelines

boolean role

The result of the **defined** operator has a boolean role.

?? expression shall be parenthesized

The rationale for using the parenthesized form of the **sizeof** operator is not applicable to the **defined** operator. However, existing practice is to use the parenthesized form (see Table 1872.1). The most common usage of the **defined** operator, in the .c files, is equivalent to using the **#ifdef** preprocessing directive.

1884 #ifdef

Example

```
#define X Y

#if defined(X) /* Checks that X is defined, not Y */
#endif
```

Table 1872.1: Occurrence of controlling expressions containing the **defined** operator (as a percentage of all **#if** and **#elif** preprocessing directives). The **#elif** preprocessing directive was followed by the **defined** operator in 66.5% of occurrences of that preprocessing directive— in the .c files (.h 75.5%). Based on the visible form of the .c and .h files.

Preprocessing Directive	%
#if defined (identifier)	15.7
#if defined (identifier) || defined (identifier)	5.8
#if defined (identifier) && defined (identifier)	2.0
#if ! defined (identifier)	1.9
#elif defined (identifier)	1.9
#if defined (identifier) && ! defined (identifier)	1.3
#if ! defined (identifier) && ! defined (identifier)	0.9
#if defined (identifier) || defined (identifier) || defined (identifier)	0.8
#if defined identifier || defined identifier	0.5
#if ! defined (identifier) && ! defined (identifier) && ! defined (identifier)	0.3
others	5.3

1873 Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token (6.4).

Commentary

preprocessing token shall have lexical form

This requirement duplicates one given elsewhere.

This sentence was added by the response to DR #304.

1874 141) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names— there simply are no keywords, enumeration constants, etc.

footnote 141

Commentary

preprocessing token converted to token

Preprocessing tokens are converted to tokens during translation phase 7.

Coding Guidelines

1878 #if identifier replaced by 0

A common mistake made by developers is to attempt to use the **sizeof** operator within a controlling constant expression. Once the **sizeof** is replaced by 0 the resulting expression is very likely to contain a syntax violation (i.e., consideration of a guideline recommendation is unnecessary because the syntax violation will cause a diagnostic to be generated).

Semantics

Preprocessing directives of the forms 1875

```
# if   constant-expression new-line group_opt

# elif constant-expression new-line group_opt
```

check whether the controlling constant expression evaluates to nonzero.

Commentary

Unlike the controlling expression of an **if** statement it is not necessary to enclose the expression in parentheses. Also the **elif** form does not exist in C language. There is no **#switch** to handle a sequence of **#if**/**#else** or **#elif** preprocessor directives (although such a construct has been proposed as a worthwhile extension, the C Committee continues to turn it down).

Rationale **#elif** was added to minimize the stacking of **#endif** directives in multi-way conditionals.

Other Languages

Some languages (e.g., Algol 68 and Ada) also include support for the **elif** keyword.

Common Implementations

concept analysis

Snelting[2] applied concept analysis to conditional inclusion directives to produce a visualization of the structure and properties of possible program build configurations.

Coding Guidelines

selection statement syntax

The coding guideline issues that are applicable to the **if** statement are also applicable to conditional inclusion directives.

Experience suggests that developers do not always use the **#elif** form in contexts where it could be used. The following are possible reasons for this behavior:

- Developers having a mental model of conditionals that does not include this construct (an equivalent construct is not available within an **if** statement).
- Developers treating **#if**/**#endif** pairs as conceptually independent of any preprocessing directives they may be nested within.
- The result of cut-and-paste operations from other parts of the source.
- Portability. One common reason why source code contains many nested conditional directives is to handle the behavior of many different translators and environments that is has been ported to. Given that historically some translators have not supported the **#elif** directive, of necessity the source may have had any use of this directive removed.

At the time of this writing there is insufficient information available to make a cost/benefit decision on a guideline recommendation and none is given here.

form of representation mixing

The *constant-expression* in a **#if** directive is sometimes evaluated by readers of the source. The expression may contain integer constants represented using a variety of different forms (e.g., decimal, hexadecimal, character constant, etc.).

A study by Gonzalaz and Kolers[1] investigated how subject's performance on an arithmetic problem was affected by the notational system used to represent values. Subjects were shown an equation of the form $p + q = n$ and had to respond true/false as quickly as possible (e.g., does $2 + 3 = 6$?). The numeric values were presented using either Arabic or Roman digits (e.g., $V + 3 = IX$, $4 + II = 6$, etc.). The results were not consistent with a model that translated the numerals into a single representation system before adding and comparing them. Others[3] have suggested that the difference in performance can be explained by the time taken to encode the different representations.

Mixed notation can occur in C source. For instance, an expression can contain integer constants represented using decimal, hexadecimal, and octal lexical forms. The results of these studies show that there is a cognitive cost associated with mixing different representations in the same expression. There is no obvious guideline recommendation for trading off this cost against the benefit of using different representations (for different expressions).

Usage

The visible form of the `.c` files contained 12,277 (`.h` 4,159) **`#else`** directives.

Table 1875.1: Common **`#if`** preprocessing directive controlling expressions (as a percentage of all **`#if`** directives). Where *`integer-constant`* is an integer constant expression, and *function-call* is an invocation of a function-like macro. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
identifier	26.5
`integer-constant`	20.3
defined (identifier)	16.4
defined (identifier) || defined (identifier)	6.0
identifier == identifier	2.4
identifier > *`integer-constant`*	2.4
identifier >= function-call	2.1
defined (identifier) && defined (identifier)	2.0
! defined (identifier)	2.0
defined (identifier) && ! defined (identifier)	1.3
identifier >= *`integer-constant`*	1.3
identifier != *`integer-constant`*	1.1
identifier < function-call	1.1
! identifier	1.1
others	14.0

Table 1875.2: Common **`#elif`** preprocessing directive controlling expressions (as a percentage of all **`#elif`** directives). Where *`integer-constant`* is an integer constant expression, and *function-call* is a function-like macro. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
defined (identifier)	49.7
identifier == identifier	19.4
defined identifier	6.6
defined (identifier) || defined (identifier)	5.7
identifier	4.7
defined (identifier) && defined (identifier)	2.6
identifier == *`integer-constant`*	1.9
identifier >= function-call	1.2
defined (identifier) || defined (identifier) || defined (identifier)	1.2
identifier >= *`integer-constant`*	1.0
others	6.1

1876 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling

#if macros expanded

constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text.

Commentary

The following quote from DR #258 has had some lines and quotes (from the C Standard) removed.

#258 ***Problem***

Consider the code:

```
#define repeat(x) x && x    // Line 1
#if repeat(defined fred)    // Line 2
```

and the code:

```
#define forget(x) 0         // Line 3
#if forget(defined fred)    // Line 4
```

Does line 2 "generate" a **defined** *operator? Is line 4 strictly conforming code, or does the fact that macro expansion "forgets" the* **defined** *operator cause a problem ?*

I would guess that the original intention was that any `defined` *X pair in the original source worked correctly. The proposed change would resolve this.*

In addition, given the order of events, it is unsuitable to say that a `defined` *X expression is "evaluated". Rather it should be described as a textual substitution.*

Proposed Committee Response

The standard does not clearly specify what happens in this case, so portable programs should not use these sorts of constructs.

If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. 1877

Commentary

Some implementations support this form and there is existing code that makes use of it. However, the C committee did not want to require this usage to be supported. The behavior is also undefined if the **defined** operator is the subject of a **#define** directive.

predefined macros not #defined

Common Implementations

Implementations vary in the order in which they process the sequence of preprocessing tokens in the constant expression. Some perform a pass that handles any **defined** operators, while others perform all processing (e.g., including macro replacement) in a single pass.

implementation single pass

Coding Guidelines

Intentional use of this behavior is likely to involve a directive that is either successfully translated or causes a diagnostic to be issued. This largely removes one rationale for a guideline recommendation. Occurrences of uses of this behavior (which may involve developers needing to comprehend what is occurring) are considered to be rare. Which removes the other rationale for a guideline recommendation.

Example

```
#if DEF
#define FOO defined
#else
#define FOO 1 +
#endif

#if FOO BAR
#endif

#define PAREN_PLUS ) + 1

#if defined(X PAREN_PLUS
#endif

#define M(a) defined(a)

#if M(X)
#endif
```

1878 After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number **0**, and then each preprocessing token is converted into a token.

#if identifier replaced by 0

Commentary

Requiring that all identifiers appearing in this context be defined as a macro has the potential for introducing a great deal of complexity. Zero has several useful arithmetic properties that enable it to occur with no, or a canceling, affect. Being able to use it as an implicit initial value, for identifiers that may be defined as macros, reduces the number of possibilities that developers need to consider.

preprocessing token converted to token

The conversion of a preprocessing token to a token mirrors that which occurs in translation phase 7. The sequencing rules mean that any remaining identifiers are converted to the pp-number **0** before token conversion occurs.

The wording was changed by the response to DR #305.

C++

In the C++ Standard **true** and **false** are not identifiers (macro names), they are literals:

. . . , except for **true** *and* **false***, are replaced with the pp-number 0, . . .* 16.1p4

If the character sequence true is not defined as a macro and appears within the constant-expression of a conditional inclusion directive, when preprocessed by a C++ translator this character sequence will be treated as having the value one, not zero.

Coding Guidelines

The analogy might be made between an identifier that has not been defined as a macro and an object that has not been explicitly initialized. However, use of this analogy requires a choice to be made, (1) for static storage duration an implicit value of zero is provided, and (2) while for automatic and allocated storage duration the value of the object is indeterminate. If the usage was unintentional, it is a fault and considered to be outside the scope of these coding guidelines. An intentional usage may cause subsequent readers to spend more time deducing that the affect of the usage is to produce the value zero, than if they had been able to find a definition that explicitly specified a zero value.

guidelines not faults

The simplest way of adhering to a guideline recommending that all identifiers appearing in the controlling expression of a conditional inclusion directive be defined would be to insert the following (replacing X by the undefined identifier) on the lines before it:

```
#ifndef X
#define X 0
#endif
```

However, given these potential usage patterns there does not appear to be a worthwhile benefit in a guideline recommendation covering this issue.

Example

In the following the developer may be expecting M to be replaced by the body of some previously defined macro. If no such macro exists M will be replaced by 0.

```
#if M == 2
#endif
```

Usage

Approximately 15% of all conditional inclusion directives, in the translated form of this book's benchmark programs, contained an identifier that was replaced by 0 (i.e., they contained an identifier that was neither the operand of **defined** or defined as macro names).

The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6~~, except that all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header **<stdint.h>**~~. 1879

Commentary

The wording was changed by the response to DR #265.

#if operand type uintmax_*

For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header **<stdint.h>**.[142)] 1880

Commentary

The preprocessor has to evaluate constant expressions using one or more integer type representations. Limiting the representations used to integer types having the same rank minimizes the number of dependencies between the preprocessor and the translator (this requirement also creates a dependency between the processor and the contents of the header **<stdint.h>** used by the implementation).

The types intmax_t and uintmax_t are the signed or unsigned version of the greatest-width integer type supported by the implementation (and have rank at least equal to that of the type **long long**). Any differences in the representation the types intmax_t and uintmax_t used by the preprocessor and the typedef names defined (if any, through the appropriate **#include**) in a translation unit would mean that an implementation was not conforming (the situation is the same as that between the type of **sizeof** and the size_t used from **<stddef.h>**, see DR #017q7).

The wording was changed by the response to DR #265.

C90

*The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 6.4 using arithmetic that has at least the ranges specified in 5.2.4.2, except that **int** and **unsigned int** act as if they have the same representation as, respectively, **long** and **unsigned long**.*

The ranks of the integer types used for the operands of the controlling constant expression differ between C90 and C99 (although in both cases the rank is the largest that an implementation is required to support). Those cases where the value of the operand exceeded the representable range in C90 (invariably resulting in the value wrapping) are likely to generate a very large value in C99.

C++

The C++ Standard specifies the same behavior as C90 (see the C90 subsection above).

Common Implementations

While a number of C90 implementations supported the type **long long** they still performed preprocessor arithmetic using the types **long** and **unsigned long** (as specified by the C90 requirements). Fear of breaking existing source code means that vendors are likely to offer some form of C90 compatibility option that will continue to perform preprocessor arithmetic using the types specified in C90.

Coding Guidelines

There are a number of possible coding guideline issues associated with the value of a constant expression in a **#if** directive, including:

- The value may be different from the value of an identical sequence of tokens in other contexts in the source file (e.g., the right operand of an assignment statement).
- The value may depend on the implementation used (this problem is not preprocessor-specific, the representation used for the operands in an expression can depend on the implementation).
- The specification has changed between C90 and C99.

The problem with any guideline recommendation is that the total cost is likely to be greater than the total benefit (a cost is likely to be incurred in many cases and a benefit obtained in very few cases). For this reason no recommendation is made here. The discussion on suffixed integer constants is also applicable in the context of a conditional inclusion directive.

integer constant
type first in list

Example

In the following the developer may assume that unwanted higher bits in the value of `C` will be truncated when shifted left.

```
#define C 0x1100u
#define INT_BITS 32

#define TOP_BYTE (C << (INT_BITS-8))

#if TOP_BYTE == 0
/* ... */
#endif

void f(void)
{
if (TOP_BYTE == 0)
   /* ... */ ;
}
```

1881 This includes interpreting character constants, which may involve converting escape sequences into execution character set members.

#if
escape sequences

Commentary

This conversion also occurs in translation phase 5.

translation phase
5

1882 Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.[143)]

Commentary

The C committee recognized that developers may choose to perform different phases of translation on different hosts. For instance, source files may be preprocessed and then distributed for further translation on other, different, hosts.

Common Implementations

EBCDIC

Differences between the numeric values in these two cases is rare (although cases involving Ascii and EBCDIC character sets do occur).

Coding Guidelines

representation information using ?? representation information using ??

Making use of the numeric value of character constants is making use of representation information, which is covered by a guideline recommendation. However, there are cases where deviations may occur.

Example

footnote 141 1874

See footnote 141.

---

basic character set may be negative

Also, whether a single-character character constant may have a negative value is implementation-defined. 1883

Commentary

basic character set positive if stored in char object

The guarantee on the value being nonnegative does not apply during preprocessing. For instance, a preprocessing using the EBCDIC character set and acting as if the type **char** was signed. In other contexts the value of a character constant containing a single-character that is not a member of the basic execution character set is implementation-defined.

character constant more than one character

Coding Guidelines

character constant more than one character

The discussion on the possibility of character constants having other implementation-defined values is applicable here.

#ifdef #ifndef

Preprocessing directives of the forms 1884

```
# ifdef  identifier new-line groupopt
# ifndef identifier new-line groupopt
```

check whether the identifier is or is not currently defined as a macro name.

Commentary

There is no **#elifdef** form (although over half of the uses of the **#elif** directive are followed by a single instance of the **defined** operator— Table 1872.1).

Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively. 1885

Commentary

The **#ifdef** and **#ifndef** forms are rather like the unary **++** and **--** operators in that they provide a short hand notation for commonly used functionality.

Coding Guidelines

The **#ifdef** forms are the most common form of conditional inclusion directive. Measurements (see Table 1872.1) also show that nearly a third of the uses of the **defined** operator could be replaced by one of these forms. There are advantages (e.g., most common form suggests most practiced form for readers, and ease of visual scanning down the left edge of the source) and disadvantages (e.g., requires more effort to add additional conditions to the single test being made) to using the **#ifdef** forms, instead of the **defined** operator. However, there does not appear to be a worthwhile cost/benefit to recommending one of the possibilities.

---

142) Thus on an implementation where INT_MAX is 0x7FFF and UINT_MAX is 0xFFFF, the constant 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7. 1886

Commentary

The wording was changed by the response to DR #265.

1887 143) Thus, the constant expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

footnote 143

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

Commentary

This situation could occur, for instance, if the Ascii representation were used during the preprocessing phases and EBCDIC were used during translation phase 5.

translation phase 5

1888 Each directive's condition is checked in order.

Commentary

The order is from the lowest line number to the highest line number.

Coding Guidelines

It may be possible to obtain some translation time performance advantage (at least for the original developer) by appropriately ordering the directives. Unlike developer behavior with **if** statements, developers do not usually aim to optimize speed of translation when deciding how to order conditional inclusion directives (experience suggests that developers often simply append new directive to the end of any existing directives).

selection statement syntax

Recognizing a known pattern in a sequence of directives has several benefits for readers. They can make use of any previous deductions they have made on how to interpret the directives and what they represent, and the usage highlights common dependencies in the source. In the following code fragment more reader effort is required to spot similarities in the sequence that directives are checked than if both sequences of directives had occurred in the same order.

```
#ifdef MACHINE_A
/* ... */
#else
#ifdef MACHINE_B
/* ... */
#endif
#endif

#ifdef MACHINE_B
/* ... */
#else
#ifdef MACHINE_A
/* ... */
#endif
#endif
```

Given the lack of attention from developers on the relative ordering of directives and the benefits of using the same ordering, where possible, a guideline recommendation appears worthwhile. However, a guideline recommendation needs to be automatically enforceable and determining when two sequences of directives have the same affect, during translation, may be infeasible because information that is not contained within the source may be required (e.g., dependencies between macro names that are likely to be defined via translator command line options).

guideline recommendation enforceable

Rev 1888.1 Where possible the visual order of evaluation of expressions within different sequences of nested conditional inclusion directives shall be the same.

If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; 1889

Commentary

if statement operand compare against 0

A parallel can be drawn with the behavior of **if** statements, in that if their controlling expression evaluates to zero, during program execution, any statements in the associated block are skipped.

directive processing while skipping

directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; 1890

Commentary

The preprocessor operates on a representation of the source written by the developer, not translated machine code. As such it needs to perform some processing on its input to be able to deduce when to stop skipping. Directives need to be processed to keep track of the level of nesting of conditionals and translation phases 1–3 still need to be performed (line splicing could affect what is or is not the start of a line) and characters within a comment must not be treated as directives.

translation phase 1

The intent of only requiring a minimum of directive processing, while skipping, is to enable partially written source code to be skipped and to allow preprocessors to optimize their performance in this special case, speeding up the rate at which the input is processed.

Example

```
#if 1
extern int ei;

#elif " an unmatched quote character, undefined behavior

extern int foo_bar;
#endif

#if 0
printf("\
#endif \n");

#endif

#if 0
```

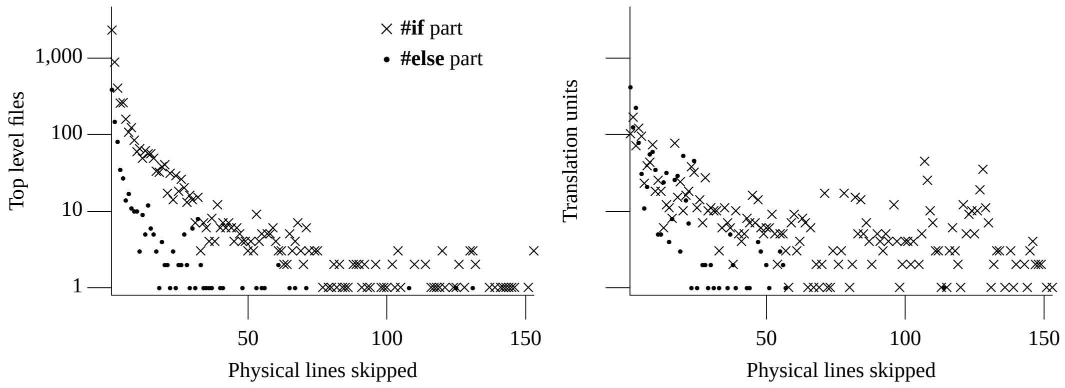

Figure 1889.1: Number of top-level source files (i.e., the contents of any included files are not counted) and (right) complete translation units (including the contents of any files **#include**d more than once) having a given number of lines skipped during translation of this book's benchmark programs.

```
/*
#endif
*/
#endif
```

1891 the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group.

Commentary

There is no requirement that any directive be properly formed, according to the preprocessor syntax. However, preprocessing tokens still need to be created, before they are ignored (as part of translation phase 3).

preprocessor directives syntax
translation phase 3

Example

In the following the **#define** directive is not well formed. But because this group is being skipped the translator is required to ignore this fact.

```
#if 0
#define M(e
#endif
```

1892 Only the first group whose control condition evaluates to true (nonzero) is processed.

Commentary

This group is processed exactly as-if it appeared in the source outside of any group.

1893 If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed;

Commentary

A semantic rule to associate **#else** with the lexically nearest preceding **#if** (or similar form) directive, like the one given for **if** statements, is not needed because conditional inclusion is terminated by a **#endif** directive.

else binds to nearest if

Like the matching **#if** (or similar form) directive case, all preprocessing tokens in the group are treated as if they appeared outside of any conditional inclusion directive. Processing continues until the first **#endif** is encountered (which must match the opening directive).

Coding Guidelines

The arguments made for **if** statements always containing an **else** arm might be thought to also apply to conditional inclusion. However, the presence of a matching **#endif** directive reduces the likelihood that readers will confuse which preprocessing directive any **#else** associates with (although other issues, such as lack of indentation or a large number of source lines between directives can make it difficult to visually associate matching directives).

else

1894 lacking a **#else** directive, all the groups until the **#endif** are skipped.[144)]

Commentary

The affect of this specification mimics the behavior of **if** statements.

else binds to nearest if

1895 **Forward references:** macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.18.1.5).

References

1. E. G. Gonzalaz and P. A. Kolers. Mental manipulation of arithmetic symbols. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8(4):308–319, 1982.
2. G. Snelting. Reengineering of configurations based on mathematical concept analysis. Technical Report Informatik-Bericht Nr. 95-02, Techniesche Universität Braunschweig, Jan. 1995.
3. I. van Rooij. Notation specificity in numerical cognition: A critique of Noël and Seron's (1992) reappraisal of the Gonzalez and Kolers (1982) study. 2000.