The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

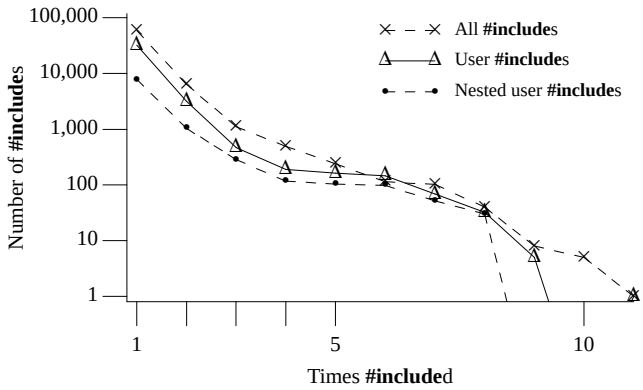

Figure 1896.1: Number of times the same header was **#include**d during the translation of a single translation unit. The crosses denote all headers (i.e., all systems headers are counted), triangles denote all headers delimited by quotes (i.e., likely to be user defined headers) and bullets denote all quote delimited headers **#include** nested at least three levels deep. Based on the translated form of this book's benchmark programs.

6.10.2 Source file inclusion

Constraints

source file inclusion

A **#include** directive shall identify a header or source file that can be processed by the implementation. 1896

Commentary

footnote 153

There is no requirement that a header be represented using a source file. It could be represented using prebuilt information within the translator that is enabled only when the appropriate **#include** directive is encountered during preprocessing (but not in a group that is skipped). Also there is no requirement that the spelling of the header in the C source file be represented by a source file of the same spelling. The C Standard has no explicit knowledge of file systems and is silent on the issue of directory structures. Minimum required limits on the implementation processing of a header name are specified elsewhere.

#include 1909 mapping to host file

Failure to locate a header or source file that can be processed by the implementation (e.g., a file of the specified name does not exist, at least along the places searched) is a constraint violation.

Other Languages

Most languages do not specify a **#include** mechanism, although many of their implementations provide one. The approach commonly used by C implementations is popular, but not universal. Some languages explicitly state that a **#include** directive denotes a file of the given name in the translators host environment.

Common Implementations

For most implementations the header name maps to a file name of the same spelling. It is quite common for the translation environment to ignore the case of alphabetic letters (e.g., MS-DOS and early versions of Microsoft Windows), or to limit the number of significant characters in the file name denoted by a header name (the remaining characters being ignored). Use of the / character in specifying a full path to a file is sufficiently common usage that even host environments where this character is not normally associated with a directory separator support such usage in header names (many Microsoft windows translators support this character, as well as the \ character, as a directory separator).

source file representation
header precompiled

In the majority of implementations **#include** directives specify files containing source in text form. However, some implementations support what are known as *precompiled headers*.

It is not uncommon (over 10% of **#include**s in Figure 1896.1) for the same header to be **#include**d more than once when translating a source file (it is a requirement that implementations support this usage for standard headers). The following are some of the techniques implementations use to reduce the overhead of subsequent **#include**s.

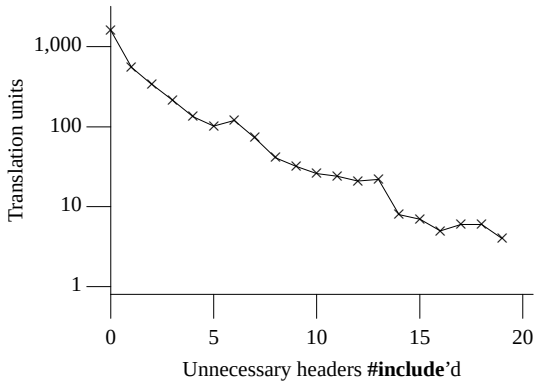

Figure 1896.2: Number of preprocessing translation units (excluding system headers) containing a given number of **#include**s whose contents are not referenced during translation (excludes the case where the same header is **#include**d more than once, see Figure 1896.1). Based on the translated form of this book's benchmark programs.

- A common convention is to bracket the contents of a header, starting with the preprocessing token sequence `#ifndef __H_file_name__/#define __H_file_name__` and ending with `#endif`. The processing of subsequent **#include**s of the same header is then reduced to the minimal processing needed to skip to the matching **#endif**. Some implementations (e.g., `gcc`) go one step further and detect headers that contain such bracketing the first time they are processed, and completely skips opening and processing the header if it is subsequently encountered again in a **#include** directive.
- Support the preprocessing directive **#import**.[1] This directive is equivalent to the **#include** directive except that if the specified header has already been included it is not included again.

Coding Guidelines

Some coding guideline documents recommend that implementation supplied headers appear before developer written headers, in a source file. Such recommendations overlook the possibility that a developer written header might itself **#include** an implementation header.

Experience suggests that once a **#include** directive appears in a source file it is rarely removed (see Figure 1896.2) and that new **#include** directives are simply added after the last one. The issue of redundant code is discussed elsewhere.

redundant code

There does not appear to be a worthwhile benefit in ordering **#include** directives in any way (apart from any relative ordering dictated by dependencies between headers).

Table 1896.1: Occurrence of two forms of *header-name*s (as a percentage of all **#include** directives), the percentage of each kind that specifies a path to the header file, and number of absolute paths specified. Based on the visible form of the `.c` files.

Header Form	% Occurrence	% Uses Path	Number Absolute Paths
<h-char-sequence>	75.0	86.4	0
"q-char-sequence"	25.0	17.2	0

Semantics

1897 A preprocessing directive of the form

#include
h-char-sequence

```
# include <h-char-sequence> new-line
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header.

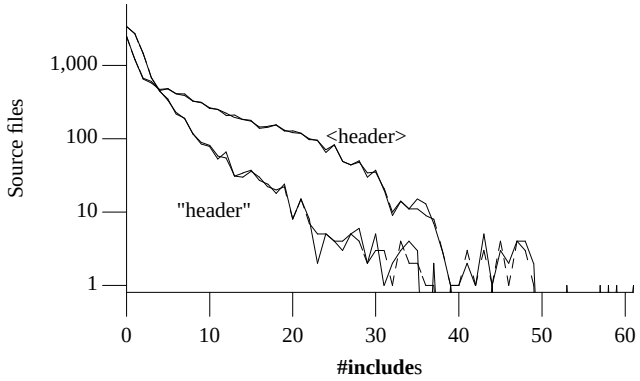

Figure 1896.3: Number of `.c` source files containing a given number of **`#include`** directives (dashed lines represent number of unique headers). Based on the visible form of the `.c` files.

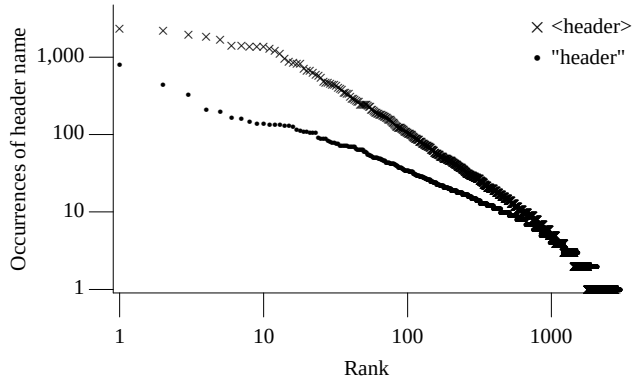

Figure 1896.4: *header-name* rank (based on character sequences appearing in **`#include`** directives) plotted against the number of occurrences of each character sequence. Also see Figure **??**. Based on the visible form of the `.c` files.

Commentary

File systems invariably provide a unique method of identifying every file they contain (e.g., a *full path name*). The base document recognized the disadvantages of requiring that the full path name be specified in each **`#include`** directive and permitted a substring of it to be given. The implementation-defined places are usually additional character sequences (e.g., directory names) added to the *h-char-sequence* in an attempt to create a full path name that refers to an existing file.

header name syntax

Rationale

The file search rules used for the filename in the **`#include`** directive were left as implementation-defined. The Standard intends that the rules which are eventually provided by the implementor correspond as closely as possible to the original K&R rules. The primary reason that explicit rules were not included in the Standard is the infeasibility of describing a portable file system structure. It was considered unacceptable to include UNIX-like directory rules due to significant differences between this structure and other popular commercial file system structures.

Nested include files raise an issue of interpreting the file search rules. In UNIX C a **`#include`** directive found within an included file entails a search for the named file relative to the file system *directory* that holds the outer **`#include`**. Other implementations, including the earlier UNIX C described in K&R, always search relative to the same *current directory*. The C89 Committee decided in principle in favor of K&R approach, but was unable to provide explicit search rules as explained above.

Other Languages

Other languages (or an extension provided by their implementations) commonly use the double-quote delimited form.

Common Implementations

The character sequence between the < and > delimiters is invariably treated as the name of a file, possibly including a path. The ordering of the search sequence used for directives having the form *<h-char-sequence>* is often different from that used for the form *"q-char-sequence"*. For instance, in the *<h-char-sequence>* case the contents of `/usr/include` might be searched first, followed by the contents of the directory containing the `.c` file, while in *"q-char-sequence"* case the contents of the directory containing the `.c` file might be searched first, followed by other places.

1909 #include mapping to host file

The environment in which a translator executes may also affect the sequence of places that are searched. For instance, the affect of relative path names (e.g., `../proj/abc.h`) on the identity of the current directory.

`gcc` searches two directories, `/usr/include` and another directory that holds very machine specific files, such as `stdarg.h` (e.g., `/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include` on your authors computer). `gcc` supports the **`#include_next`** directive. This directive causes the search algorithm to skip some of the initial implementation-defined places that would normally be searched. The initial places that are skipped are those that were searched in locating the file containing the **`#include_next`** directive (including the place where the search succeeded).

Tzerpos and Holt[2] describe a well-formedness theory of header inclusion that enables unnecessary **`#include`** directives to be deduced.

Coding Guidelines

The standard does not specify the order in which the implementation-defined places are searched. This is a potential coding guideline issue because it is possible that a *h-char-sequence* will match in more than one of the places (i.e., there is a file having the same name along several of the different possible search paths). The behavior is thus dependent (i.e., it is assumed that the contents of the different headers will be different) on the order in which the places are searched.

Experience suggests that the affect of a translator locating an **`#include`**d file different from the one expected to be located by the developer has one of two consequences— (1) when the contents of the file accessed is similar to the one intended (e.g., a different version of the intended file) the source file may be successfully translated, and (2) when the contents of the file accessed has no connection with the intended file the source is rarely successfully translated. The problem might therefore be considered to be one of version management, rather than the choice of characters used in a *h-char-sequence*. There are a number of reasons why a solution to this issue is to not use *h-char-sequence*s at all, including the following:

1898 #include places to search for

- For the < > delimited form, implementations usually look in a predefined location first (as described in the Common implementation section above and in the following C sentence). Ensuring that the names chosen by developers for the headers they create are different from those of system headers is an almost impossible task. While it might be possible to enumerate the set of names of existing file names of system headers contained in commercially important environments, members are likely to be added to this set on a regular basis. Rather than trying to avoid using file names likely to match those of system headers, developers could ensure that places containing system headers are searched last.

- The < > delimited form is often considered to denote *externally* supplied headers (e.g., provided by the implementation or translator environment vendor). What constitutes a system supplied header is open to interpretation. One distinction that can be made between system and developer headers is that developers do not control of the contents of system headers. Consequently, it can be argued that their contents are not subject to coding guidelines. Headers whose contents have been written by developers are subject to coding guidelines. The convention generally adopted to indicate this status is to use the double-quote character delimit form of **`#include`**.

Rev 1897.1 Developer written headers in a **#include** directive shall not be delimited by the < and > characters.

Developers sometimes specify full path names in headers (see Table 1896.1). This is a configuration management issue and is not considered to be within the scope these coding guidelines.

Table 1897.1: Number of various kinds of identifiers declared in the headers contained in the `/usr/include` directory of some translation environments. Information was automatically extracted and represents an approximate lower bound. Versions of the translation environments from approximately the same year (mid 1990s) were used. The counts for ISO C assumes that the minimum set of required identifiers are declared and excludes the type generic macros.

Information	Linux 2.0	AIX on RS/6000	HP/UX 9	SunOS 4	Solaris 2	ISO C
Number of headers	2,006	1,514	1,264	987	1,495	24
macro definitions	10,252	18,637	13,314	11,987	10,903	446
identifiers with external linkage	1,672	1,542	1,935	616	1,281	487
identifiers with internal linkage	80	34	2012	0	5	0
tag declaration	716	1,088	899	1,208	945	3
typedef name declared	1,024	828	15	493	1,027	55

#include places to search for

How the places are specified or the header identified is implementation-defined. 1898

Commentary

The differences between the environments in which translation occurs has narrowed over the years. However, even although there may be much common practice, such are issues are considered to be outside the scope of the C Standard.

program transformation mechanism

Common Implementations

Implementations invariably search one or more predefined locations first (e.g., `/usr/include`), followed by a list of alternative places. A number of techniques are used to allow developers to specify a list of alternative places to be searched for files corresponding to the headers specified in a **#include** directive. For instance, the alternative places may be specified via a translator command line option (e.g., `-I`), in a translator configuration file (e.g., `gcc` version 2.91.66 hosted on RedHat Linux reads many default from the file `/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs`, although the path `/usr/include` is still hard coded in the translator sources), or an environment variable (e.g., several Microsoft windows based translators use `INCLUDE`).

The directory separator used in Unix and MS-DOS slants in different directions. Many implementations, in both environments, recognize both characters as directory delimiters. One consequence of this is that escape sequences are not recognized as such (something that is unlikely to be a problem in header names).

The RISCOS environment does not support filenames ending in `.h`. The implementation-defined behavior for this host is to look in a directory called `h`, for a file of the given name with the `.h` removed.

Coding Guidelines

The implementation-defined behavior associated with how the places are specified occurs outside of the source code and is the remit of configuration management guidelines. For this reason nothing further is said here.

#include q-char-sequence

A preprocessing directive of the form 1899

 # include "*q-char-sequence*" *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters.

Commentary

The commonly accepted intent of this form of the **#include** directive is that it is used to reference source files created by developers (i.e., headers that are not provided as part of the implementation or host environment). The only syntactic difference between *q-char-sequence* and *h-char-sequence* is that neither sequence may contain their respective delimiters.

header name syntax

Most *q-char-sequence*s end with one of two character sequences (i.e., `.c` or `.h`). The character sequences before these suffixes is often called the *header name*.

Other Languages

The use of double-quote as the delimiter is the almost universal form used in other languages (although some use the `'` character because that is what is used to delimit string literals).

Coding Guidelines

The term commonly used to refer to these source files is *header*. The context of the conversation often being used to distinguish any other intended usage. The intent is that the contents of these source files is controlled by developers and as such they are subject to coding guidelines.

1900 The named source file is searched for in an implementation-defined manner.

Commentary

While this "implementation-defined manner" might be the same as that for the < > delimited form. The intent is for it to be sufficiently different that developers do not need to be concerned about the name of a header created by them matching one provided as part of the implementation (and therefore potentially found by the translator when searching for a matching header). For instance, your author does not know the names of most of the 304 files (e.g., `compface.h`) contained in `/usr/include` on his software development computer. The discussion on the < > delimited form is applicable here.

1897 #include h-char-sequence

Common Implementations

The search algorithm used invariably differs from that used for the < > delimited form (otherwise there would be little point in distinguishing the two cases). The search algorithm used by some implementations is to first look in the directory containing the source file currently being translated (which may itself have been included). If that search fails, and the current source file has itself been included, the directory containing the source file that **#include** it is then searched. This process continuing back through any nested **#include** directives. For instance, in:

file_1.c
```
#include "abc.h"
```

file_2.c
```
#include "/foo/file_1.c"
```

file_3.c
```
#include "/another/path/file_2.c"
```

(assuming the translation environment supports the path names used), translating the source file `file_3.c` causes `file_2.c` to be included, which in turn includes `file_3.c`. The source file `abc.h` will be searched for in the directories `/foo`, `/another/path` and then the directory containing `file_3.c`.

Some implementations use the double-quote delimited form within their system headers, to change the default first location that is searched. For instance, a third-party API may contain the header `abc.h`, which in turn needs to include `ayx.h`. Using the form `"ayx.h"` means that the implementation will search in the directory containing `abc.h` first, not `/usr/include`. This usage can help localize the files that belong to specific APIs. Other implementations use a search algorithm that starts with the directory containing the original source file being translated.

#include 1898 places to search for

If the source file is not found after these places have been searched, some implementations then search other places specified via any translator options. Other implementations simply follow the behavior described by the following C sentence (which has the consequence of eventually checking these other places).

If this search is not supported, or if the search fails, the directive is reprocessed as if it read 1901

```
# include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.

Commentary

The previous search can fail in the sense that it does not find a matching source file.

Some existing code uses the double-quote delimited form of **#include** directive to include headers provided by the implementation (rather than the < > delimited form). This requirement ensures that such code continues to be conforming.

---

footnote 144

144) As indicated by the syntax, a preprocessing token shall not follow a **#else** or **#endif** directive before the terminating new-line character. 1902

Commentary

Saying in words what is specified in the syntax.

Common Implementations

Many early implementations (and some present days ones, for compatibility with existing source) treated any sequence of characters following one of these directives as a comment, e.g., `#endif X == 1`.

However, comments may appear anywhere in a source file, including within a preprocessing directive. 1903

Commentary

comment replaced by space

A comment is replaced by a single space character prior to preprocessing.

preprocessing directive ended by

A preprocessing directive of the form 1904

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted.

Commentary

#include 1914 example 2

This form permits the < > or double-quote delimited forms to be generated via macro expansion. However, it is rarely used (11 instances in over 60,000 **#include** directives in the visible source of the `.c` files). Whether this is because developers are unaware of its existence, or because it has little utility is not known.

#include macros expanded

The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) 1905

Commentary

To be exact, the preprocessing tokens after **include** in the directive up to the first new-line character are processed just as in normal text.

(Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) 1906

Commentary

This C sentence provides explicitly clarification that macro replacement occurs in this case (the same clarification is also given elsewhere).

#line macros expanded

1907 The directive resulting after all replacements shall match one of the two previous forms.[145)]

Commentary

It is not a violation of syntax if the directive does not match one of the two previous forms, because the syntax of this form has been matched. It is a violation of semantics and therefore the behavior is undefined.

1908 The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

Commentary

This implementation-defined behavior may take a number of forms, including:

- The **##** operator can be used to glue preprocessing tokens together. However, the behavior is undefined if the resulting character sequence is not a valid preprocessing token. For instance, the five preprocessing tokens {{} {string} {.} {h} {}} cannot be glued together to form a valid preprocessing token without going through intermediate stages whose behavior is undefined. ## operator ## if result not valid
- Creating a preprocessing token, via macro expansion, having the double-quote delimited form (i.e., a string preprocessing token) need not depend on any implementation-defined behavior. The stringize operator can be used to create a string preprocessing token. # operator
- Other implementation-defined behaviors might include the handling of space characters. For instance, in the following:

```
#define bra <
#define ket >
#include bra stdio.h ket
```

  does the implementation strip off the space character at the ends of the delimited character sequence?

Coding Guidelines

Given the rarity of use of this form of **#include** no guideline recommendations are given here.

Example

```
#define mk_sys_hdr(name) < ## name ## >

#if BUG_FIX
#define VERSION 2a /* works because pp-numbers include alphabetics */
#else
#define VERSION 2
#endif

#define add_quotes(a) # a
#define mk_str(str, ver) add_quotes(str ## ver)

#include mk_str(Version, VERSION)
```

1909 The implementation shall provide unique mappings for sequences consisting of one or more ~~letters or digits (as defined in 5.2.1)~~ nondigits or digits (6.4.2.1) followed by a period (.) and a single ~~letter~~ nondigit. #include mapping to host file

Commentary

This C sentence and the following ones in this C paragraph are a specification of the minimum set of requirements that an implementation must meet. For sequences outside of this set the implementation mapping may be non-unique (like, for instance, the Microsoft Windows technique of mapping files ending in `.html` to `.htm`). The handling of character sequences that resemble UCNs may also differ, e.g., `"\ubada\file.txt"` (Ubada is a city in Tanzania and BADA is the Hangul symbol 붚 in ISO 10646). The standard does not permit any number of period characters because many operating systems do not permit them (at least one, RISCOS, does not permit any).

The wording was changed by the response to DR #302 to extend the specification to be more consistent with C++.

C++

16.2p5 *The implementation provides unique mappings for sequences consisting of one or more nondigits (2.10) followed by a period (.) and a single nondigit.*

Other Languages

Other languages either specified to operate within the same operating systems and file systems limitations as C and as such have to deal with the same issues, or require an integrated development environment to be created before they can be used.

Common Implementations

Implementations invariably pass the sequence of characters that appear between the delimiters (when searching other places a directory path may be added) as an argument in a call to `fopen` or equivalent system function. The called library function will eventually call some host operating system function that interfaces to the host file system. The C translator's behavior is thus controlled by the characteristics of the host file system and how it maps character sequences to file names. The handling of the period character varies between file systems, known behaviors include:

- Unix based file systems permit more than one period in a file name.
- MS-DOS based file systems only permit a single period in a file name.
- RISCOS, an operating system for the Acorn ARM processor does not support filenames that contain a period. For this host file names, that contained a period, specified in a **`#include`** directive were mapped using a directory structure. All file names ending in the characters `.h` were searched for in a directory called h.

Coding Guidelines

Because an implementation is not required to provide a unique mapping for all sequences it is possible that an unintended header or source file will be accessed, or the translator will fail to identify a known header or source file. The possible consequences of an unintended access are discussed elsewhere, while failure to identify known header or source file will cause a diagnostic to be issued. The cost/benefit issues associated with using character sequences having a unique mapping in the different environments that the source may be translated in is outside the scope of these coding guidelines.

#include 1897 h-char-sequence
source file 1896 inclusion

The first character shall ~~be a letter~~ not be a digit. 1910

Commentary

This requirement only applies to the first character of the sequence that implementations are required to provide a unique mapping for.

The wording was changed by the response to DR #302.

C90

The requirement that the first character not be a digit is new in C99. Given that it is more restrictive than that required for existing C90 implementations (and thus existing code) it is unlikely that existing code will be affected by this requirement.

C++

This requirement is new in C99 and is not specified in the C++ Standard (the argument given in the C90 subsection (above) also applies to C++).

Common Implementations

Most implementations support a first character that is not a letter.

1911 The implementation may ignore the distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

header name significant characters

Commentary

These permissions reflect known characteristics of file systems in which translators are executed.

C90

The limit specified by the C90 Standard was six significant characters. However, implementations invariably used the number of significant characters available in the host file system (i.e., they do not artificially limit the number of significant characters). It is unlikely that a header of source file will fail to be identified because of a difference in what used to be a non-significant character.

C++

The C++ Standard does not give implementations any permissions to restrict the number of significant characters before the period (16.1p5). However, the limits of the file system used during translation are likely to be the same for both C and C++ implementations and consequently no difference is listed here.

Common Implementations

All file systems place some limits on the number of characters in a source file name— for instance:

- Most versions of the Microsoft DOS environment ignore the distinction of alphabetic case and restrict the mapping to eight significant characters before any period (and a maximum of three after it).
- POSIX requires that at least 14 characters be significant in a file name (it also requires implementations to support at least 255 characters in a pathname). Many Linux file systems support up to 255 characters in a filename and 4095 characters in a pathname.

Coding Guidelines

The potential problems associated with limits on sequences characters that are likely to be treated as unique is a configuration management issue that is outside the scope of these coding guidelines.

1912 A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

Commentary

Thus **#include** directives can be nested within source files whose contents have themselves been **#include**d. This issue is discussed elsewhere. While this permission only applies to source files, an implementation using some form of precompiled headers (which are not source files within the standard's definition of the term) that did not support this functionality would not be popular with developers.

limit #include nesting
header precompiled
source files

1913 EXAMPLE 1 The most common uses of **#include** preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

Other Languages

Some languages only have a single form of **#include** directive for all headers.

#include example 2

EXAMPLE 2 This illustrates macro-replaced **#include** directives: 1914

```
#if VERSION == 1
        #define INCFILE  "vers1.h"
#elif VERSION == 2
        #define INCFILE  "vers2.h"   // and so on
#else
        #define INCFILE  "versN.h"
#endif
#include INCFILE
```

Commentary

This example does not illustrate any benefit compared to that obtained from placing separate **#include** directives in each arm of the conditional inclusion directive.

Forward references: macro replacement (6.10.3). 1915

---

footnote 145

145) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); 1916

Commentary

translation phase 6

String concatenation occurs in translation phase 6 and so it is not possible to join together two existing strings to form another string within a **#include** directive.

---

thus, an expansion that results in two string literals is an invalid directive. 1917

Commentary

It is an invalid directive in that it violates a semantic requirement and thus the behavior is undefined. It is not a syntax violation.

References

1. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.

2. V. Tzerpos and R. C. Holt. A well-formedness theory of C source inclusion. In *Proceedings of the* 3^{rd} *International Symposium on Applied Corporate Computing (ISACC'95)*, pages 18–22, Oct. 1995.