The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.4.9 Comments

Commentary

Comments do not affect the behavior of a program. They are intended to be of use to developers or tools that read the source code containing them. A number of tools use comments to help them do their job. These usually work by specifying sequences of characters at the start of the comment, which act as flags that can be detected when the tool reads the source. Uses include the following:

- *Version control.* Some revision control tools, such as `rcs`, search for identification keywords while checking out source. Matches against such keywords (e.g., `$Revision$` or `$Date$`) cause the keyword to be replaced with the appropriate values.[1] A source file may contain such identification keywords in comments, or sometimes in string literals (e.g., `static const char rcsid[] = "$Header$";`). The advantage of string literals is that they appear in the translated object code.
- *Documentation extraction.* Special flag characters indicate how the remaining source text is to be treated (e.g., man page, TEX source, etc.[8]).

Other Languages

In some implementations of Basic, mostly interpreter-based and seen in early hobbiest computers, the program source code was held in storage and interpreted directly. Comments needed to be skipped during program execution. In such environments the presence of comments could adversely affect a programs performance.

Common Implementations

There have been a few implementations that allowed translator directives to be placed within comments. The majority of modern translators use the **`#pragma`** preprocessing directive for this purpose.

#pragma directive

Coding Guidelines

Many coding guideline documents[14, 16] specify what they consider to be good commenting practices. Although the use of comments is generally considered to be a *good thing* to do, there have been no studies quantifying their costs or benefits. These coding guidelines do not get involved in making recommendations on how to comment. There are a number of reasons for this:

- Writing effective comments is a skill (these coding guidelines do not aim to teach skills).
- Automatic enforcement of any guideline recommendation dealing with comments is likely to be difficult. If some form of commenting were shown to have a worthwhile cost/benefit ratio, the corresponding recommendation would need to be handled through code reviews. (The state of the art in static analysis of comments is many years away from having the natural language semantics capability needed for automatic enforcement.)

The grammar of comment layout

Technically comments can appear between any two preprocessing tokens in the source. In practice comments invariably appear in only a small number of locations in the source, relative to other preprocessing tokens. Experience shows that most developers visually organize comments; there might be said to be a comment layout grammar. Whether the use of a comment grammar simply represents the original author's desire for a pleasing layout, is simply a by-product of following simple rules while writing code, or is a cost-effective mechanism for reducing the effort needed by subsequent readers to comprehend the source is not known.

The presence of comments can also affect the layout of the constructs to which they refer. Components of this comment grammar include:

- Comments *attached* to the source statement they refer to (e.g., visually located immediately above the source line it refers to, or to the right of it on the same line).

- Vertical alignment of comment boundaries. Visual neatness as an aid in associating one comment spread over multiple lines.
- *header* comments (e.g., at the start of a source file, or function definition) provide information about the sequence of lines/statements that follow it).

```
1  if (valu == 0) /* Comment to the right of the if it refers to. */
2     {
3     blah blah ; /* Comment about one statement. */
4
5  if (valu == 0) /* Comment about the if.               */
6     {           /* Continuing commentary about the if. */
7     blah blah ; /* and even more continuation of general commentary. */
8
9  /* Comment refers to statement below it. */
10 x=3;
11
12 /* Previous blank line helps delimit this comment from about C statement. */
13 y=3;
14
15 /*
16  * We could write a long comment that describes the conditions
17  * under which x and y take on certain values.  Developers are
18  * likely to associate the x & y mentioned in this comment with the
19  * identifier names in the if statement below.
20  */
21 if ((x == 3) && /* This comment implicitly refers to x. */
22     (y == 4))   /* Expression layout has been integrated into comment structure. */
```

Overview

Writing comments is a cost that has no short-term benefit for the developer who incurs that cost. They are purely an investment for a possible future benefit, probably to a different developer than the one who wrote them. Comments can reduce, and sometimes increase, the cost of comprehending source code.

Comments can reduce the effort needed to comprehend source code by providing information in a form that requires less effort to comprehend than source code. Comments can also increase the probability that subsequent modifications are correct by providing background information (i.e., on intended affects) that is not explicitly contained in the adjacent source code.

Comments can increase the effort needed to comprehend source code by providing incorrect, or misleading, information. The presence of comments can also increase the effort needed to visually scan source code. The visual organization of comments often has a structure that is separate from the grammar of the surrounding statements and declarations.[18]

Comments are used for a variety of purposes, including:

- *To contain management information.* This can include the change history of a file, or function, and the person responsible. It might also be cross-references to other source files and documents, the comments effectively providing a means of embedding links within a source file.
- *To present information in a diagramatic fashion* (the hope being that this alternative form of presentation will be easier for the reader to interpret than source code).
- *To present information in natural language prose.* For readers new to the source code, the use of natural language may provide a more direct route to a reader's model of the world than the code itself.
- *To create place markers in the source.* For instance, placing a comment on the closing brace of a compound block to indicate the kind of statement that occurs at the start of the block (if, while, switch, etc.). Another example is the specification of source yet to be written at that point.

Documentation in comments

The quantity of documentation associated with programs can vary enormously. The case of no documentation, outside of the source code, is not uncommon, while for some large projects a large percentage of the total effort went into creating documentation.[2] Having this kind of information within the same file as the source code has several advantages:

- If the source code is available, the documentation is available (developers will be familiar with programs whose documentation has been lost, or requires significant effort to obtain).
- The colocation of documentation and source code might be thought to ensure that both are updated together (experience suggests that this is not always the case).

cost/accuracy trade-off

- People perform a cost/accuracy calculation when deciding where to obtain information from. Having documentation available in the source file reduces information access cost (compared to having to obtain it from another file), potentially leading to increased accuracy of the information used.

comment disadvantages

It also has disadvantages:

- Information within comments, which duplicates what appears in other documents, runs the risk of not being updated when the original document is changed, or vice versa. Duplicating information creates the problem of keeping both up-to-date, and if they are differences between them, knowing which is correct.

belief maintenance

- Opinions formed early, based on limited data, interfere with information presented later using more accurate data.

The environment in which developers work can also have an impact. An environment that provides support services for ensuring that documentation is maintained and readily available may have less of a need to duplicate large amounts of information in comments. In an environment where support services are poor and developers are responsible for providing and maintaining their own documentation, there are plenty of reasons why this task may not get done. When support services are poor, having documentation within source code may be the only way of ensuring that this information is available to subsequent maintainers.

literate programming

Literate programming

An extreme form of colocating source code and documentation is espoused by the so-called *literate programming*[11] approach to documentation. Here the source code and its associated documentation are kept in a single file. Various tools are used to separate out the program source and its documentation. The development method proposed by Knuth has gained some influential supporters and needs to be discussed. The choice of the term *literate programming* expresses an admirable intent. However, by concentrating on the end result— a beautifully laid out, typeset program and associated documentation— Knuth has completely ignored the context where most of a developer's interaction with source code occurs (i.e., reading the original source). The original *literate programming* source from which both the source code and program documentation are extracted is much more difficult to read than either of the two documents it contains.

The target market for Knuth's form of literate programming is widespread publication of source and documentation, where it is expected to be read by many people, usually for educational or training purposes. Comprehension of the contents of the original material (in its single file) only need be performed by a small number of people, and is not required to be simple or easy to do. Much of the published praise of Knuth's literate programming has concentrated on the quality of the final output documents. There is no published research comparing the costs of maintaining two separate documents versus a single document from which both the code and documentation is extracted. Your author can see no obvious advantages to using Knuth's literal programming approach to software development in a commercial environment. In this environment there are a small readership of the code and documentation, and most of the time is spent working with this material directly.

Reducing source code comprehension costs

The effectiveness of a comment has to be measured by the extent to which it reduces the cost of developer comprehension of the surrounding source code. Writing effective comments is a different kind of skill than writing code. Developers receive feedback when they write code, at the very least the translator issues diagnostics if they violate a constraint or syntax rule. The only way developers receive feedback on the effectiveness of their comments is when other people read them. In the short-term, the only time when this is likely to occur is during code review. The following are some of the cost/benefit analysis issues that apply to the use of comments:

- *Practice and feedback are needed to learn to write effective comments.* Is the cost of learning to comment, including the cost of poor commenting that will have been written in source during that period, significantly less than the benefits likely to be obtained later? The issue of who pays this cost also needs to be considered in light of the probability of trained developers changing jobs.
- *Comments may need to be updated when the source they refer to is modified.* Does the person making the source code change know enough to be able to make an associated, meaningful change to the comment? What is the cost associated with updating all of the related comments? As source code nears the end of its useful life, it may be more cost effective to delete comments when the source they refer to is modified rather than updating them. (The update cost is not likely to be recouped; deleting comments removes the potential liability caused by them being incorrect.)
- *The liabilities of incorrect, or out-of-date, comments.* If the contents of comments disagrees with the behavior of the source code developers become confused and need to spend extra time deducing which is correct. This deduction process may reach an incorrect conclusion, possibly leading to faults being introduced.
- *How much benefit do comments provide?* There has been no study, using experienced developers, that has attempted to measure the benefit of comments.

A study by Roediger, Jacoby, and McDermott[15] looked at the false memories created by misleading information. They found that memories of past events are influenced by previous recollections of those events. Also information retrieved during the most recent account of an event may have a larger effect on the current recollection than the original event itself.

Self commenting code

There is a school of thought that claims it is possible to write self-documenting programs, rendering comment redundant. It is just a matter of choosing the right names for identifiers. The function

```
1  unsigned int square_root(unsigned int valu)
2  {
3  unsigned int root = 0;
4
5  valu = (valu + 1) >> 1;
6  while (root < valu)
7     valu -= root++;
8
9  return root;
10 }
```

contains no comments, but its name makes it obvious to readers what it does. Is that sufficient? Are we going to trust that this algorithm really does return the square root of its argument? A comment giving a brief outline of the mathematics behind the algorithm might increase confidence in its behavior. IEC 60559 requires that implementation support a square root operation. A comment could explain that for certain ranges of argument this function is faster than making use of hardware-supported square root (which requires conversions to/from floating point, plus the actual square root operation).

A study of maintenance programmers by IBM[6] found that understanding the original authors intent was the most difficult problem they had.

Comment layout

Experience shows that developers do not like writing comments. Making it more difficult than it is already perceived to be could result in less time being spent in creating comments. Some developers invest a lot of time in formatting their comments. Whether or not this cost leads to any measurable benefit is debatable. However, some of the layouts used can require additional workload (compared to alternative layouts) should they need to be modified. For instance, in the comment

```
1  /*********************************************************
2   * Allocate a local variable of the given size.          *
3   * The offset depends on which way the system stack grows.*
4   * (Although the offsets are always positive)            *
5   *                                                       *
6   *  If the stack is ascending, we need to                *
7   *     i. Align on the correct boundary                  *
8   *    ii. The offset of the variable is the current offset*
9   *   iii. Increment the offset to allow for this variable *
10  *                                                       *
11  *  If the stack is descending, we                       *
12  *     i. Increment the current offset for this variable  *
13  *    ii. Align on the correct boundary                  *
14  *   iii. The offset of the variable is the current offset*
15  *                                                       *
16  * If the stack is descending, the offsets from the frame *
17  * pointer (fp) are negative within the interpreter but   *
18  * stored as positive offsets in mcc.  Thus aligning      *
19  * upwards on mcc's offsets actually aligns downwards in  *
20  * memory (which is what we require).                     *
21  *********************************************************/
```

the bounding of the text by star characters may be visually appealing (leaving aside the issue of whether any comment is important enough to warrant this degree of visual impact). But maintaining this layout, when the comment is updated, requires additional developer effort. Reducing the expected cost of writing a comment may increase the likelihood that no updates will be made to the contents of existing comments.

```
1  /*********************************************************
2     Lines can be added to this comment without the need to worry
3     about overrunning the length of the box that contains them,
4     or having to use tab/space as padding to add a terminating *
5
6     Existing wording can be edited without having to spend time
7     repositioning all those * characters.
8
9     The contained text is still visibly delimited.
10  *********************************************************/
```

Comment layout is a complex process[18] and even the simplest of worthwhile recommendations are likely to be very difficult to enforce automatically (interested readers can find suggestions in other coding guideline documents[14, 16]). The following recommendation is intended to help ensure that existing comments are kept up-to-date.

Rev .1 Comments shall not be laid out in a fashion that requires significant additional effort for developers wanting to modify their contents.

Visual affects

The presence of comments in source can create visual patterns that distract developers' attention away from patterns that may present in the declarations and statements. In the following example, the aligned comments down the right have the effect of creating two vertical visual groupings. The prominence of the horizontal empty space, separating declarations from statements, is reduced (see Figure **??**).

```
1  #define BASE_COST 23             /* Production tooling costs.           */
2
3  extern int widgets_in_product;
4
5  int sum_widget_costs(void)       /* Return how much this will cost.     */
6  {
7  int num_widgets;                 /* A local count of widgets.           */
8  int total_cost;                  /* Running total of the costs.         */
9                                   /* this is the value returned.         */
10 widgets_in_product--;             /* Subtract one to make it zero-based. */
11
12 total_cost=BASE_COST;             /* A startup cost cannot be avoided.   */
13 if (widgets_in_product != 0)     /* make sure we have something to do.  */
14    { /* ... */ }
15 }
```

By drawing attention to itself, this form of blocked commenting has taken attention away from the declarations and statements. Comments have the lowest attention priority and their presentation needs to reflect this fact. Straker[16] discusses visual effect issues in more detail.

Example

```
1  extern int *ptr;
2
3  void f(void)
4  {
5  int loc = 4 /*ptr; /* Blah blah. */;
6  }
```

Usage

While over 30% of the characters in this book's benchmark programs (see Table **??**) are contained within comments, they only represent around 2% of the tokens. A study by Fluri et al[7] of the releases of three large Java programs over a 6 year period (on average) found three different patterns in the ratio of number of comment lines to number of non-comment lines for each program.

A study of comments in C++ source by Etzkorn[5] found that 57% contained English sentences (that could be automatically parsed by the tool used).

Table .1: Common formats of nonsentence style comments. Adapted from Etzkorn, Bowen, and Davis.[5]

Style of Comment	Example
Item name— Definition	MaxLength— Maximum CFG Depth.
Definition	Maximum CFG Depth.
Unattached prepositional phrase	To support scrolling text.
Value definitions	0 = not selected, 1 = is selected.
Mathematical formulas	Can be Boolean expressions...

Table .2: Breakdown of comments containing parsable sentences. Adapted from Etzkorn, Bowen, and Davis.[5]

Percentage	Style of Sentence	Example
51	Operational description	This routine reads the data. Then it opens the file.
44	Definition	General Matrix— rectangular matrix class.
2	Description of definition	This defines a NIL value for a list.
3	Instructions to reader	See the header at the top of the file.

Comments are usually written in the present tense, with either indicative mood or imperative mood.

Table .3: Common formats of sentence-style comments. Adapted from Etzkorn, Bowen, and Davis.[5]

Part of Speech	Percentage	Example
Present Tense	75	
Indicative mood, active voice		This routine reads the data.
Indicative mood, active voice, missing subject		Reads the data.
Imperative mood, active voice		Read the data.
Indicative mood, passive voice		This is done by reading the data.
Indicative mood, passive voice, missing subject		Is done by reading the data.
Past Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine opened the file. or Opened the file.
Future Tense	4	
Indicative mood, either active or passive voice, occasional missing subject		This routine will open the file. or Will open the file.
Other	15	

comment /*

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. 934

Commentary

The exception cases are implied by the phases of translation.

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

Other Languages

All programming languages support some form of commenting. The character sequences used to introduce the start of a comment vary enormously; for instance, Basic uses the character sequence **`rem`**, while Scheme and many assembly languages use **`;`**. Java supports what is called a documentation comment: such a comment is introduced by the character sequence `/**`; the additional `*` character distinguishing it from a nondocumentation comment.

Coding Guidelines

This form of comment introduction enables the creation of multiline comments. This has the advantage of simplifying the job of writing a long comment and the disadvantage of comments sometimes terminating in unexpected locations in the source.

commenting out

Developers sometimes use the `/*` style of commenting to *comment out* sections of source to prevent it being translated and executed. This might be necessary, for instance, because the source code is not yet working properly, or because it has been decided that its functionality is not needed at the moment. Use of comments for this purpose is sometimes said to run the risk of having a nested comment change the intended

effect. In practice, the change of behavior usually occurs during translation, not program execution; for instance, attempting to comment out the following sequence of statements

```
some_value = 3;
if (test_bit == 1)
  total++; /* Special case. */
do_something();
```

will cause a syntax violation at the closing comment sequence placed after the call to `do_something`. The opening comment sequence placed before the assignment to `some_value` being terminated by the `*/` characters appearing in the comment within the source being commented out.

Although it might be thought that a syntax violation would be sufficient warning for the developer to look more closely at the source, experience suggests that developers can become confused to the extent of deleting the terminating comment characters without deleting the introductory comment characters (the syntax violation goes away). Using the `/*` form of comments to temporarily stop declarations or statements influencing the behavior of a program is a known root cause of faults.

Some implementations provide an option to support the nesting of comments. The standard explicitly states that comments do not nest. If it is necessary to prevent source code from being translated, either the **`#if`** preprocessing directive (because such directives nest), or the `//` form of commenting needs to be used. 939 footnote 70 936 comment //

Cg 934.1 The `/*` kind of comment shall not be used to comment out source code.

This guideline recommendation does not prevent the use of source code within comments where it is plainly intended to be part of the exposition of the comment— for instance:

```
/*
 * A discussion on some application issue and the following is an
 * example of one possible way of solving it:
 *
 * some_value = 3;
 * if (test_bit == 1)
 *   total++;
 * do_something();
 *
 * But we chose not to do it this way ...
 */
```

One way of distinguishing commented out code from code that is part of a comment is to examine the context and commenting style. A simple opening comment character sequence, followed by declarations or statements, followed by the closing comment character sequence is obviously commented out code. In the above case the presence of comment text that discusses the code might be viewed as sufficient to distinguish the usage (of course it could have been a comment that happened to precede the statements that was subsequently merged with the following comment). The use of star characters at the start of every line further reduces the probability that this is commented out code.

A practical way of measuring the probability of source code within a comment being commented out code is the ease with which it can be converted to executable machine code. Only having to remove a single line, at the start and end of the comment, gives a high probability of the source being commented out. In the preceding case it is necessary to delete the first four lines and last three lines of the comment, followed by deleting the star character at the start of every line. It is very unlikely to be commented out source.

When using the `/*` form of commenting, care has to be taken to ensure that the contents of the comment are easily distinguishable from what is outside the comment. The following example shows how poor layout might lead to confusion:

```
extern void farm(int *);

void f(void)
{
int loc;

/* We are now going to perform some calculation that involves
a complicated formula.  The details can be found in
barns (1999), which is the best reference */
farm(&loc);
}
```

There are a number of techniques developers can use to make comments and their contents appear as a distinct visual unit. Starting each line of comment with a character that rarely occurs at the beginning of a noncomment line creates continuity. Making it easy for readers to match the opening and closing comment character sequences helps to create visual symmetry.

```
extern void farm(int *);

void f(void)
{
int loc;

/*
 * We are now going to perform some calculation that involves
 * a complicated formula.  The details can be found in
 * barns (1999), which is the best reference
 */
farm(&loc);
}
```

sentence-picture relationships

Comments sometimes contain a diagram. A theoretical discussion of the advantages of a diagram over a purely sentence-based description is given by Larkin and Simon.[12] Experimental verification that pictures can enhance text memory is provided by a study by Waddill and McDaniel.[10] Readers of the source often compare diagrams against sequences of statements in the source code. The intent of this comparison is to verify that the two representations are consistent with each other. The following discussion is based on studies by Clark and Chase,[4] and Carpenter and Just.[3]

In a study by Clark and Chase[4] subjects were shown a display consisting of a sentence and a picture. They had to quickly press a button indicating whether the sentence was true or false. The sentences were "star is above plus", "star is below plus", "star isn't above plus", and "star isn't below plus", and the same four sentences with the words *star* and *plus* swapped. The pictures had the form $\overset{*}{+}$, or $\overset{+}{*}$. These sentences and pictures can be combined into four different kinds of questions. The sentence could be true/false, they could also be affirmative (state that a relationship is true) or negative (state that a relationship is not true) (see Table 934.1).

Table 934.1: Four types of questions.

Statement Relative to Fact	Example
true-affirmative (TA)	star is above plus: $\overset{*}{+}$
false-affirmative (FA)	plus is above star: $\overset{+}{*}$
false-negative (FN)	star isn't above plus: $\overset{*}{+}$
true-negative (TN)	plus isn't above star: $\overset{*}{+}$

Clark and Chase created a model using four parameters (whether above/below was used, the sentence being true/false, the sentence being stated in a negative form was the sum of two parameters) to account for

the differing delays in subjects' responses to the information displayed. The predicted response time for answering a question could be obtained by adding the delays required by the parameters.

Carpenter and Just built a model (known as *The Constituent Comparison Model*; each proposition within the mental representation is referred to as a constituent) that combined these four parameters into one. This model makes two assumptions:

1. The information content of sentences and pictures is assumed to be mentally represented in propositional form. A proposition can be affirmative or negative; for instance, "The star is above the plus" is represented as *(AFFIRMATIVE (ABOVE, STAR, PLUS))*, and "The star isn't above the plus" as *(NEGATIVE (ABOVE, STAR, PLUS))*. Propositions can be embedded within one another *e.g., {FORTUNATE [NEG (RED, DOTS)]}). Pictures are assumed to always be represented affirmatively; that is, the mental representation specifies what the picture is, not what it is not.

propositional form

2. The comparison process, between the two propositional forms, uses a single mental operation—retrieve and compare. Corresponding constituents from the sentence and picture representations are retrieved and compared, pair by pair. A subject's delay in responding is determined by the number of these operations.

The algorithm for determining the number of retrieve and compare operations is:

A boolean flag is used to hold the result state of the comparison process; its initial value is assumed to be true. Every time a mismatch is encountered the flag changes state (it can flip-flop between them). The time required for the change-of-state operation is assumed to be small, compared to the retrieve-and-compare operation.

When the comparison of a constituent proposition mismatches, the following operations occur:

1. the flag changes state,
2. the mismatching constituent is tagged to indicate that when the restarted comparison process encounters it again, it should be treated as a match,
3. the comparison process goes back to the innermost constituents and starts comparing from where it first started.

The time taken to determine whether a sentence matches a picture is proportional to the number of comparison operations. Two consequences of the Just and Carpenter model are that the greater the number of mismatches, the greater the number of comparison operations needed, and those mismatches that occur later will require more comparison operations than those that occur earlier. These predictions are borne out by the response timing from a variety of studies.[4]

This model assumes that pictures are represented propositionally. Is this always the case? A study by MacLeod, Hunt, and Mathews[13] found that 23% of subjects maintained a visual representation of the picture and converted the sentences they read into a mental image of the picture described. Because these subjects used pictorial representations, the linguistic structure of the sentence (e.g., the use of a negative) could not affect their performance.

Example

```
1  /* A simple comment on a single line, so why was this style used? */
2
3  #define OBSCURE_TEN (2/*/*/*/*/*/5)
4
5  /*
6   * A comment with a simple, straight-forward, easy-to-understand
7   * format.  Hmmm, if *'s appear at the start of a line then why not // ?
8   */
```

Table 934.2: Occurrence of kinds of comments (as a percentage of all comments; last row as a percentage of all new-line characters). Based on the visible form of the `.c` and `.h` files.

Kind of Comment	`.c` files	`.h` files
/* comment */	91.0	90.1
// comment	9.0	9.9
/* on one line */	70.3	79.1
new-lines in /* comments	12.3	17.5

comment contents only examined to

The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[70)]

Commentary

translation phase 3 trigraph sequences phase 1 line splicing

Comments are processed in translation phase 3. Trigraphs and line splicing will already have been handled.

C++

The C++ Standard gives no explicit meaning to any sequences of characters within a comment. It does call out the fact that comments do not nest and that the character sequence `//` is treated like any other character sequence within such a comment.

2.7p1 *The characters `/*` start a comment, which terminates with the characters `*/`.*

Other Languages

Some languages support the use of translator directives within comments. This directive might control, for instance, the generation of listing files, the alignment of storage, and the use of extensions. Java supports the use of HTML tags inside its documentation comments. A few languages (e.g., Common Lisp) support nested comments and the contents of this form of comment need to be examined to identify the start/end of each nested comment.

Common Implementations

The contents of comments are sometimes examined by tools that analyze source code like a translator. The SVR5 `lint` tool[17] includes the following:

```
1  extern int select;
2
3  void f(void)
4  {
5  if (select == 3)
6     {
7     return;
8     /* NOTREACHED */  // Indicate that we know the statement is not reached
9     select--;
10    }
11 switch (select)
12    {
13    case 3: select++;
14    /* FALLTHRU */  // Indicate that we know the case falls through
15    case 4: select++;
16            break;
17    }
18 }
```

Coding Guidelines

Skipping all characters until the sequence `*/` is found is a rather open-ended operation. Experience suggests that the `/* */` style of comment is not ideally suited for single-line comments; for instance, developers sometimes omit the closing `*/` characters. Also the characters `*/` occupy two positions on a line, which sometimes causes lines to wrap on display devices. The `//` form of comments does not have these problems.

Example

```
1  /\
2  * line splicing occurs before the comment is recognized
3  *\
4  /
5
6  /* ??) is a trigraph.  More importantly so is *??/
7  /
```

---

936 Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character.

comment //

Commentary

Many comments occupy a single line. The only form of comment supported by C90 required an explicit end-of-comment delimiter. Experience showed that omission of this closing delimiter is a cause of program faults. Also source code editors often wrap long lines and the two characters needed to close a comment can generate the need to abbreviate a comment or to have to accept a wrapped line in the displayed source. This newly introduced form of commenting does not suffer from these problems.

The C preprocessor is often used as a general preprocessor for other languages. The introduction of // as a comment start sequence in C99 is likely to cause problems for some developers; for instance, Fortran format statements can contain sequences of these two characters.

C90

Support for this style of comment is new in C99.

There are a few cases where a program's behavior will be altered by support for this style of commenting:

```
1  x = a //* */ b
2        + c;
3
4  #define f(x) #x
5
6  f(a//) + g(
7  );
```

Occurrences of these constructs are likely to be rare.

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

Other Languages

This style of comment is supported in BCPL, C++, Java, and many languages created in the last 10 years. Ada and Basic support the same commenting concept, they terminate at the end of line; the character sequences being used are `--` and **REM**, respectively. Fortran has always supported this concept of commenting; source lines that contain the letter C in the fifth column are treated as comments.

Common Implementations

Many implementations supported this style of commenting in their C90 translators.

---

937 The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

Commentary

As previously pointed out, this statement is a simplification.

935 comment contents only examined to

C++

The C++ Standard includes some restrictions on the characters that can occur after the characters //, which are not in C90.

2.7p1 *The characters // start a comment, which terminates with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required.*

A C source file using the // style of comments may use form-feed or vertical-tab characters within that comment. Such a source file may not be acceptable to a C++ implementation. Occurrences of these characters within a comment are likely to be unusual.

Coding Guidelines

The // comment form is expected to occupy a single physical source line. It often comes as a surprise to readers of the source if such a comment includes the line that follows it.

Cg 937.1 The physical line containing the // from of comment shall not end in a line splice character.

EXAMPLE 938

```
"a//b"                  // four-character string literal
#include "//e"          // undefined behavior
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();          // syntax error, not comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

Commentary

This example lists some of the more visually confusing character sequences that can occur in source code.

footnote 70 70) Thus, /* ... */ comments do not nest. 939

Commentary

translation phase 3 A comment is a single lexical entity, replaced by one space character in translation phase 3. Support for nested comments would require that they have an internal structure.

Other Languages

While few language specifications support the nesting of comments, some of their implementations do.

Common Implementations

Some pre-C Standard translators supported the nesting of comments and some translators[9] continue to provide an option to support this functionality (although support for such an option is not as common as for many other pre-C Standard behaviors).

Coding Guidelines

The /* comment form is sometimes used for *commenting out* source code. Such usage is error prone because comments might already occur within the source that is intended to be removed from subsequent translator processing (meaning that some of this source is not *commented out*). This issue is discussed elsewhere. Enabling a translator's support for nested comments is doing more than enabling an extension, it can also change the behavior of strictly conforming programs.

934 commenting out
?? extensions cost/benefit

Example

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  if (/*/*/0*/**/1)
6     printf("On this implementation comments nest\n");
7  else
8     printf("On this implementation comments do not nest\n");
9  }
```

References

1. D. Bolinger and T. Bronson. *Applying RCS and SCCS*. O'Reilly & Associates, Inc, 1995.
2. F. P. Brooks, Jr. *The Mythical Man-Month*. Addison–Wesley, anniversary edition, 1995.
3. P. A. Carpenter and M. A. Just. Sentence comprehension: A psycholinguistic processing model of verification. *Psychological Review*, 82(1):45–73, 1975.
4. H. H. Clark and W. G. Chase. On the process of comparing sentences against pictures. *Cognitive Psychology*, 3:472–517, 1972.
5. L. H. Etzkorn, L. L. Bowen, and C. G. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(1):1–18, 1999.
6. R. K. Fjelstad and W. T. Hamlen. Applications program maintenance study: Report to our respondents. In G. Parikh and N. Zvegintzov, editors, *Tutorial on Software Maintenance*. IEEE Computer Society Press, 1979.
7. B. Fluri, M. Würsch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE)*, page ???, Oct. 2007.
8. W. Groop. Users manual for doctext: Producing documentation from C source code. Technical Report Technical Report ANL/MCS-TM-206, Argonne National Laboratory, University of Chicago, Mathematics and Computer Science Division, 1995.
9. Hitachi Ltd. *H8S, H8/300 Series C Compiler User's Manual*. Hitachi Ltd, 4.0 edition, Sept. 1998.
10. W. P. J and M. A. McDaniel. Pictorial enhancement of text memory: Limitations imposed by picture type and comprehension skill. *Memory & Cognition*, 20(5):472–482, 1992.
11. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
12. J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.
13. C. M. MacLeod, E. B. Hunt, and N. N. Matthews. Individual differences in the verification of sentence-picture relationships. *Journal of Verbal Learning and Verbal Behavior*, 17:493–507, 1978.
14. S. McConnell. *Code Complete*. Microsoft Press, 1993.
15. H. L. Roediger III, J. D. Jacoby, and K. B. McDermott. Misinformation effects in recall: Creating false memories through repeated retrieval. *Journal of Memory and Language*, 35:300–318, 1996.
16. D. Straker. *C-Style standards and guidelines*. Prentice Hall, Inc, 1992.
17. Unix System Laboratories, Inc. *Unix System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. Prentice Hall, Inc, 1990.
18. M. L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, Oct. 2002.