

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.4.6 Punctuators

punctuator  
syntax

*punctuator*: one of

```
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:~:
```

### Commentary

In most cases it is only necessary for a lexer to lookahead one character when dividing up the input stream into preprocessing tokens (the need to potentially tokenize the punctuators `%:~:` and `...` requires two characters of lookahead, for when the last character is not one needed to build them).

The only graphic characters from the Ascii character set not included in this list are `$`, `@`, and `\` (which is used to indicate an escape sequence).

### C90

Support for `<: :> <% %> %: %:~:` was added in Amendment 1 to the C90 Standard. In the C90 Standard there were separate nonterminals for punctuators and operators. The C99 Standard no longer contains a syntactic category for operators. The two nonterminals are merged in C99, except for `sizeof`, which was listed as an operator in C90.

### C++

The C++ nonterminal *preprocessing-op-or-punc* (2.12p1) also includes:

```
:: .* ->* new delete
and and_eq bitand bitor compl
not not_eq or or_eq xor xor_eq
```

The identifiers listed above are defined as macros in the header `<iso646.h>` in C. This header must be included before these identifiers are treated as having their C++ meaning.

### Other Languages

C contains a much larger set of punctuators than most other programming languages. Some languages use the `$` character as an operator/punctuator (e.g., Snobol 4). One of the original design aims of Cobol was to make programs written in it read like English prose. To this end keywords were used to represent operators that were normally indicated by punctuators in other languages:

```
1 01 data-value PIC 9(3) OCCURS 10 TIMES.
2 MOVE x to b.
3 ADD x to y GIVING z.
4 WRITE report-out AFTER ADVANCING PAGE.
```

Some languages (e.g., Ada and Fortran) include the `**` operator (usually as a binary operator denoting exponentiation). Fortran uses abbreviated names for some operators (e.g., `.EQ.` for `==` and `.LT.` for `<`).

### Common Implementations

Some implementations include the `@` character in the list of punctuators.

## Coding Guidelines

Some punctuators are visually very similar to other punctuators (e.g., != and |=). Other punctuators are very easily overlooked— for instance, the comma character. One of the reasons for these visual similarities is the font used to display source. The importance of presenting characters in an easy-to-read form has been recognized in other disciplines; for instance, chemistry and mathematics have specific fonts that have been designed for them. While a great deal of effort has been invested in creating fonts that make it easier to read text documents, such as books, or grab people’s attention (e.g., advertising) very little effort has gone into designing a font to make source code easier to read.

Baecker and Marcus<sup>[1]</sup> make a number of interesting suggestions. For instance, unary operators can be made more noticeable by having them in superscript (e.g.,  $i = j * -k$  versus  $i = j *^{-} k$  or  $x = y + + - z$  versus  $x = y^{++} - z$ ). While some fonts do deal with character pairs, the pairs that occur in source tend to be separated by other characters; for instance, matching bracketing characters, such as [ ], ( ), and { }, can be nested within each other. Having the outer brackets displayed in a larger point size (e.g., (((...))) makes the job of matching them up much easier to do visually.

It would be impractical for these coding guidelines to recommend the use of particular fonts when any that have been tuned to the display of C source are not commonly available, or where such usage requires particular tool support.

**Table 912.1:** Commonly used terms for punctuators and operators.

Punctuator/ Operator	Term	Punctuator/ Operator	Term
[ ]	left square bracket or opening square bracket or bracket	^	circumflex or xor or exclusive or
( )	left round bracket or opening round bracket or bracket or parenthesis		vertical bar or bitwise or or or
{ }	left curly bracket or opening curly bracket or bracket or brace	&&	and and or logical and
.	dot or period or full stop or dot selection		logical or or or
->	indirect or indirect selection	?	question mark
*	times or star or dereference or asterisk	:	colon
+	plus	;	semicolon
-	minus or subtract	...	dot dot dot or ellipsis
~	tilde or bitwise not	=	equal or assign
!	exclamation or shriek	==	times equal
++	plus plus	/=	divide equal
--	minus minus	%=	percent equal or remainder equal
&	and or address of or ampersand or bitwise-and	+=	plus equal
/	slash or divide or solidus	--=	minus equal
%	remainder or percent	<<=	left-shift equal
<<	left-shift	>>=	right-shift equal
>>	right-shift	&=	and equal
<	less than	^=	xor equal or exclusive or equal
>	greater than	=	or equal
<=	less than or equal	,	comma
>=	greater than or equal	#	hash or sharp or pound
==	equal	##	hash hash or sharp sharp or pound pound
!=	not equal	<: :>	no commonly used terms
		<% %> %:	
		%: %:	

**Table 912.2:** Occurrence of *punctuator* tokens (as a percentage of all tokens; multiply by 1.88 to express occurrence as a percentage of all punctuator tokens). Based on the visible form of the .c and .h files.

Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens	Punctuator	% of Tokens
,	8.82	==	0.53		0.16	--	0.03
)	8.09	:	0.46	+=	0.11	++v	0.02
(	8.09	-v	0.40	>	0.11	%	0.02
;	7.80	*p	0.40	<<	0.09	--v	0.01
=	3.08	+	0.38	?:	0.08	...	0.01
->	3.00	*v	0.34	?	0.08	>>=	0.01
}	1.87	&	0.32	=	0.08	^	0.01
{	1.87	!	0.31	>=	0.07	+v	0.00
.	1.26	v++	0.27	/	0.06	%=	0.00
*	1.10	&&	0.26	>>	0.06	##	0.00
#	1.00	!=	0.26	~	0.05	*=	0.00
]	0.96	<	0.22	v--	0.04	/=	0.00
[	0.96	-	0.19	&=	0.04	<<=	0.00
&v	0.58		0.17	<=	0.04	^=	0.00

## Semantics

A punctuator is a symbol that has independent syntactic and semantic significance.

913

### Commentary

The syntactic form of a punctuator that is a *preprocessing-token* also has the syntactic form of a punctuator that can be converted (in translation phase 6) to a *token*. Some punctuators have syntactic significance only (e.g., ;), while others can occur in several contexts— for instance, the pair ( ) can be used to bracket expressions or to denote the function call operator.

### C90

*A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value.*

The merging of this distinction between operators and punctuators, in C99, makes no practical difference.

### C++

This observation is not made in the C++ Standard.

Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts).

914

### Commentary

This defines the term *operator*. The meaning of some punctuators is dependent on the context in which they occur. For instance, the : token can occur after a case label in a bit-field declaration, or as the second operator in a ternary expression using the ? operator. (The : character can also appear as one of the characters in a digraph.)

### C90

In the C90 Standard operators were defined as a separate syntactic category, some of which shared the same spelling as some punctuators.

*An operator specifies an operation to be performed (an evaluation) that yields a value, or yields a designator, or produces a side effect, or a combination thereof.*

915 An *operand* is an entity on which an operator acts.

### Commentary

This defines the term *operand*, which is common to most languages.

### C++

The nearest C++ comes to defining *operand* is:

*An expression is a sequence of operators and operands that specifies a computation.*

5p1

916 In all aspects of the language, the six tokens<sup>67)</sup>

<: :> <% %> %: %:%:

behave, respectively, the same as the six tokens

[ ] { } # ##

except for their spelling.<sup>68)</sup>

### Commentary

The intent of introducing digraphs was to produce more readable alternatives to trigraphs. The character sequences chosen are shorter and matching pairs do contain some symmetry.

### C90

These alternative spellings for some tokens were introduced in Amendment 1 to the C90 Standard. As such there is no change in behavior between C90 and C99.

### Other Languages

Digraphs are unique to C (and C++).

### Common Implementations

Even though digraphs were first introduced in 1993, vendors have been slow to add support for them in translators.

The Perkin-Elmer C compiler<sup>[2]</sup> treated the following character sequences:

(| |) (< >) \!! \! \(\ \) \^

as being equivalent, respectively, to the tokens (the last four character pairs were also treated as escape sequences):

[ ] { } || | { } ~

escape se-  
quence  
syntax

### Coding Guidelines

Digraphs have several advantages over trigraphs. They are more readable and they are not substituted for inside string literals and character constants. Their only disadvantage is the continuing lack of support in a large number of translators. The characters used to denote digraphs were chosen for the very low probability of their occurring in existing programs outside of string literals. It is possible for a program to exhibit behavior that depends on whether digraphs are supported or not. But such usage is likely to be rare and not worth a guideline recommendation.

**Example**

```
1  #include <stdio.h>
2
3  #define prog_code(x) prog_string(%) x)
4  #define prog_string(y) "prog: " #y
5
6  void c_feather(void)
7  {
8  printf("%s\n", prog_code(10));
9  }
```

Without support for digraphs the output is:

```
prog: %: 10
```

With support for digraphs the output is:

```
prog: "10"
```

**Usage**

The visible form of the .c files contained zero digraphs.

---

**Forward references:** expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8). 917

## References

1. R. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison–Wesley, Reading, MA, USA, 1990.
2. Perkin-Elmer. *C PROGRAMMING Manual*. Perkin-Elmer Corporation, Oceanport, New Jersey 07757, 1984.