# The New C Standard (Excerpted material)

## An Economic and Cultural Commentary

**Derek M. Jones**
derek@knosof.co.uk

## 6.10.9 Pragma operator

### Semantics

_Pragma operator

A unary operator expression of the form: 2030

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting the **L** prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash.

#### Commentary

escape sequence handling #

This defines the term *destringized*. This operation the inverse of the operation performed by the stringize operator.

Rationale

As an alternative syntax for a **#pragma** directive, the **_Pragma** operator has the advantage that it can be used in a macro replacement list. If a translator is directed to produce a preprocessed version of the source file, then expressions involving the unary **_Pragma** operator and **#pragma** directives should be treated consistently in whether they are preserved and in whether macro invocations within them are expanded.

The prior art on which this directive was based comes from the Cray Standard C compiler (see WG14/N449).

#### C90

Support for the **_Pragma** unary operator is new in C99.

#### C++

Support for the **_Pragma** unary operator is new in C99 and it is not available in C++.

#### Coding Guidelines

Support for the **_Pragma** operator is new in C99 and at the time of this writing there is insufficient experience available in its use to know whether any guideline recommendation is worthwhile.

#### Example

The **_Pragma** operator can be used to reduce the visual clutter in source code. For instance, rather than duplicating a sequence of conditional inclusion directives, controlling a **pragma** directive, in the various source files or function definitions that require them, they can be encapsulated in a single macro definition.

```
#if defined(Machine_A)
   #define IZATION_HINT _Pragma("ivdep")       /* Ignore vector dependencies. */
#elif defined(Machine_B)
   #define IZATION_HINT _Pragma("independent") /* Iterations are independent. */
#endif

void N449_a(int n, double * a, double * b)
{
#if defined(Machine_A)
   #pragma ivdep            /* Vectorization hint (ignore vector dependencies).   */
#elif defined(Machine_B)
   #pragma independent      /* Parallelization hint (iterations are independent). */
#endif
while (n-- > 0)
   {
   *a++ += *b++;
   }
}

void N449_b(int n, double * a, double * b)
{
```

```
IZATION_HINT
while (n-- > 0)
   {
   *a++ += *b++;
   }
}
```

2031 The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive.

**Commentary**

The string literal, or any macro invocations used to create it, will have already been processed by translation phases 1–3. Consequently the resulting sequence of characters will not include any new-line characters (because a string literal cannot them) or trigraph sequences (they will have been replaced in translation phase 1 and the escape sequence processing that might create new trigraph sequences does not occur until translation phase 5).

string literal syntax
trigraph sequences phase 1
translation phase 5

2032 The original four preprocessing tokens in the unary operator expression are removed.

**Commentary**

Exactly when this removal occurs, within translation phase 4 (other preprocessing tokens are deleted at the end of translation phase 4), is not specified. Because defining **_Pragma** as a macro name (e.g., `#define _Pragma func_call`) results in undefined behavior it is not possible for the timing of the removal to affect the output of a strictly conforming program.

preprocessing directives deleted

2033 EXAMPLE A directive of the form:

EXAMPLE _Pragma

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING ( ..\listing.dir )
```

**Commentary**

The macro expansion sequence is:

```
LISTING ( ..\listing.dir )
PRAGMA(listing on "..\listing.h")
_Pragma("listing on \"..\\listing.h\"")
```

with the character `.` being treated as a “each non-white-space character that cannot be one of the above”.

preprocessing token syntax

## References