The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.6 Pragma directive

Semantics

#pragma directive

A preprocessing directive of the form 1994

```
# pragma pp-tokens_opt new-line
```

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)[149] causes the implementation to behave in an implementation-defined manner.

Commentary

Rationale The **#pragma** directive was added in C89 as the universal method for extending the space of directives.

The word *pragma* is from the Greek word meaning *action*. The term *compiler directive* or simply *directive* is often used to refer to **pragma** directives.

In the following example the preprocessing token **STDC** does not immediately follow **pragma** (prior to macro replacement) and implementation-defined behavior can occur.

```
1  #define GLOBAL_FP_CONTRACT STDC FP_CONTRACT ON
2
3  #pragma GLOBAL_FP_CONTRACT
```

C90

The exception for the preprocessing token **STDC** is new in C99.

C++

The exception for the preprocessing token **STDC** is new in C99 and is not specified in C++.

Other Languages

A common implementation mechanism, across all languages, is for certain sequences of characters within a comment to be given a special meaning. A few languages (e.g., Ada and Algol 68) have pragma directives defined in their specification. A larger number of language implementations provide some form of **pragma** directive (although the identifier used to denote it varies). A study by Fowler[3] examined implementation-dependent pragmas in Ada compilers. He found that:

1. fewer than 5% of the implementations support all the language-defined pragmas,
2. fewer than 30% of the implementations documented their support for language-defined pragmas,
3. fewer than 45% of the implementations gave more than a perfunctory description of implementation-defined pragmas and attributes.

Common Implementations

Many implementations include support for one or more **pragma** directive, some have extensive support.[4, 7] The commercial pressure on vendors is to support the **pragma** directives specified by the market leader in their niche. Some implementations support a preprocessing directive of the form *#pack pp-tokens new-line*. It is used to specify information about how storage allocation, for various types, is to be performed. An equivalent form using a **pragma** directive might be `#pragma pack(2)`. The OpenMP API[1] for shared-memory parallelism in C makes extensive use of **#pragma**.

It is possible for the implementation-defined behavior to replace the **pragma** directive with an executable statement. For instance:

```
1  if (cond)
2     {
3  #pragma vendor_X_library_call
4     }
```

might expand to (when using a preprocessor that recognizes the **pragma** directive):

```
1  if (cond)
2     {
3  __X_go_faster_stripes();
4     }
```

or to (when using a preprocessor that does not recognize the **pragma** directive):

```
1  if (cond)
2     {
3     }
```

Some implementations and tools embed directives in comments. For instance, Lint[6] uses the form `/*UPPERCASEKEYWORD*/`, while Splint[2] uses the form `/*@information@*/`.

Coding Guidelines

The **pragma** directive is one of several possible methods that are usually available to a developer to influence how a translator performs its job (another one is command line options). However, one difference that usually exists between **pragma** directives and other methods is that the former might be applied to part of a source file, rather than all of it. For instance, the second **pragma** in a source file may change the behavior switched on by the first directive. There are a number of potential coding guideline issues associated with **#pragma** directives, including:

- The rationale for the appearance of a **pragma** directive, in source code, is to change a translator's default behavior. Any occurrences of this directive thus require readers to remember and take into account special case information, that differs from their knowledge of the default behavior, about the constructs affected by the directive. The amount of information they will need to remember will depend on whether the directive applies to all or parts of a source file.
- The visibility of the directive to readers. Other methods of altering translator behavior may also be visibly opaque (e.g., command line options in make-files). However, readers are generally aware of the places to look for translator configuration information and while readers may look at the first few lines of a source file for translator options they tend not to look at other places in a source file.
- Cutting and pasting is a relatively common method of editing source. One of the dangers of this technique is that **#pragma** directives may be unintentionally copied.

On method of minimizing many of these problems is to place all **pragma** directives near the start of source files. However, this rather defeats the flexibility offered by this directive in being selectively applied to parts of a source file. If directives apply to complete declarations it may be worthwhile placing those declarations in a separate source file (the issues involved in deciding which declarations to put in which source file are discussed elsewhere).

declarations in which source file

There is no obvious, worthwhile, guideline recommendation that can be made about how to structure the use of **pragma** directives and so nothing more is said about the issue here.

1995 The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner.

Commentary

implementation-defined behavior

The definition of implementation-defined behavior does not include these possibilities. The permissive behaviors of the **pragma** directive are thus all those permitted by the definition of implementation-defined behavior plus those listed in this C sentence.

C90

These possibilities were not explicitly specified in the C90 Standard.

C++

These possibilities are not explicitly specified in the C++ Standard.

---

pragma unrecognized ignored

Any such **pragma** that is not recognized by the implementation is ignored. 1996

Commentary

The standard does not delimit the extent to which a pragma needs to be "not recognized" by an implementation (such questions are invariably regarded as being quality-of-implementation issues that are outside the scope of the standard).

```
1   #pragma DING DUNG /* Only valid second pp-token is DONG */
```

Other Languages

Ada specifies similar behavior.

Coding Guidelines

redundant code

A **pragma** that is not recognized by the implementation is redundant code. It may not be recognized for a number of reasons, including the following:

- It is not support by the implementation. This usage might be considered harmless.
- The developer has misspelled a directive that is supported by the implementation. This is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines not faults

The extent to which it is cost effective for the author to provide information about the likelihood of a particular **pragma** being supported by various implementations (by, for instance, using a comment or conditional inclusion) is difficult to estimate for the general case. At the time the **pragma** directive is written its author will be aware of the use of implementation-defined behavior but may not have any idea of what other implementations, if any, the source will be ported to (e.g., writers of open source often give no thought to the possibility that a translator other than `gcc` might be used). For these reasons no guideline recommendations are considered here.

If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[150] whose meanings are described elsewhere: 1997

```
#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
on-off-switch:  one of
                ON        OFF        DEFAULT
```

Commentary

The identifier **STDC** is not reserved in other contexts and a source file may define a macro name that has this spelling. The term *standard pragmas* is commonly used by members of the C committee to refer to these **pragma** directives (it also appears in a footnote).

2002 footnote 150

The behavior of these **pragma** directives are specified in the library section.

The purpose of these pragmas is to inform the implementation that during translation code appearing after an occurrence of one of them is to be treated in some special way (e.g., certain floating-point optimizations can/cannot be performed). In many implementations the floating-point environment is dynamic (i.e., it can be reconfigured during program execution). The effect of any instances of these pragmas is static (i.e., it only has any effect during translation) and it is the developers responsibility to ensure that the dynamic behavior matches that specified statically.

C90

Support for the preprocessing token **STDC** in **pragma** directives is new in C99.

C++

Support for the preprocessing token **STDC** in **pragma** directives is new in C99 and is not specified in the C++ Standard.

Other Languages

Although Ada defines 38 standard pragmas it does not use a prefix to introduce them.

Common Implementations

Some vendors and other standards specify that a specific identifier follow **#pragma**. For instance, the IBM C compiler for AIX[5] uses the identifier **ibm** and the OpenMP[1] uses **omp**.

Some processors only provide static control of floating-point modes, i.e., the rounding direction must be encoded as part of the bit pattern of a generated machine code instruction.

1998 **Forward references:** the **FP_CONTRACT** pragma (7.12.2), the **FENV_ACCESS** pragma (7.6.1), the **CX_LIMITED_RANGE** pragma (7.3.4).

1999 149) An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**).

footnote 149

Commentary

Specifying that macro replacement does not occur within standard pragmas gives the implementation the freedom to use them within the headers it supplies. A macro name, with the same spelling as a preprocessing token occurring in a standard pragma, defined before a header is included will not affect a translator's interpretation of that pragma.

C90

This footnote is new in C99.

C++

This footnote is new in C99 and is not specified in the C++ Standard.

Common Implementations

This behavior is another aspect of the implementation-defined nature of **pragma** directives. If the market leaders perform macro replacement then it is very likely that other vendors will also perform it.

Those preprocessors that support macro replacement within **pragma** directives only need to examine the preprocessing token following a `#pragma` to know whether or not to perform macro replacement.

Coding Guidelines

Given that unrecognized **pragma** directives are ignored by translators it may take developers some time before they notice a difference in implementation behavior (between the cases when macro replacement occurs and does not occur).

1996 pragma unrecognized ignored

Example

The following example illustrates one technique for effectively performing macro replacement in **pragma** directives:

```
#define wanted ON
#define not_wanted OFF

#define Pragma(...) _Pragma(#__VA_ARGS__)
#define xPragma(...) Pragma(__VA_ARGS__)

xPragma(STDC FP_CONTRACT wanted)
xPragma(STDC FENV_ACCESS not_wanted)
```

If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; 2000

Commentary

The standard only guarantees a standard **pragma** directive to be treated as such is when the specified sequence of preprocessing tokens appears in the source.

C90

Support for standard pragmas is new in C99.

C++

Support for standard pragmas is new in C99 and is not specified in the C++ Standard.

Example

```
#if HAS_FAST_FLOAT_POINT_UNIT
#define DO_CONTRACT STDC FP_CONTRACT ON
#else
#define DO_CONTRACT NULL_PRAGMA
#endif

#pragma DO_CONTRACT
```

---

an implementation is permitted to behave as if it were the standard pragma, but is not required to. 2001

Commentary

This combination of behavior is subsumed within the definition of implementation-defined behavior and is specified here to clarify that it is permitted in a conforming program.

---

footnote 150

150) See “future language directions” (6.11.8). 2002

References

1. Anon. OpenMP C and C++ application program interface. Technical Report Version 2.0, OpenMP Architecture Review Board, Mar. 2002.
2. D. Evans and D. Larochelle. *Splint Manual*. University of Virginia, 3.0.6 edition, Feb. 2002.
3. K. J. Fowler. A study of implementation-dependent pragmas and attributes in Ada. Technical Report CMU-SEI-89-SR-19, Software Engineering Institute, Carnegie Mellon University, Nov. 1989.
4. IBM. *Preprocessing Directives - #pragma*. International Business Machines, 5.0 edition, 2000.
5. IBM Canada Ltd. *C for AIX Compiler Reference*. International Business Machines Corporation, May 2002.
6. S. C. Johnson. Lint, a C program checker. Technical Report Computing Science Technical Report No.65, Bell Telephone Laboratories, 1977.
7. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.