New Form of Pragma
Version 3
WG14/N593
X3J11/96-057

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

September 5, 1996

Abstract

The pragma provides a syntax for implementation-specific extensions to C. Here an alternative syntax is proposed that allows a pragma to be included in the replacement-list of a macro definition.

1 Introduction

Note that Version 3 differs from Version 2 only in section 3. The proposed edits to the C9X Draft in that section have been revised and approved by the editorial review committee for this proposal. Text for the rationale, also approved by that committee, appears in a new subsection. The remainder of the proposal is unchanged except for the respelling of `Pragma` *as* `pragma`.

The pragma preprocessing directive can be used to specify a wide range of implementation-dependent details. In the Cray Standard C compiler, for example, these include requesting additional names for a function, specifying an alternative calling sequence for a function, and enabling vector or parallel execution of the iterations of a loop. These uses range from merely convenient to essential for acceptable performance.

The syntactic specification of the pragma as a preprocessing directive has proven to be a mixed blessing. It has the advantage of being almost completely unstructured, and that gives each implementation a lot of freedom in specifying the details of the pragmas it supports. On the other hand, this means that different implementations have different ways of specifying essentially equivalent pragmas. Programs that are meant to be portable among a number of systems, will likely need to use different pragmas on each, as shown in Figure 1.

Figure 1 *Conditionally compiled pragmas*

```
#if defined(Machine_A)
  /* Request fastest calling sequence for machine A */
# pragma fast_call
#elif defined(Machine_B)
  /* Request fastest calling sequence for machine B */
# pragma vfunction
#endif
void f(int n, double * a, double * b) {
    #if defined(Machine_B)
      /* Vectorization hint (ignore vector dependencies) */
    # pragma ivdep
    #elif defined(Machine_C)
      /* Parallelization hint (iterations are independent) */
    # pragma independent
    #endif
    while(n-- > 0) {
        *a++ += *b++;
    }
}
```

The obvious problem with such conditional inclusions of pragmas is that they can be so verbose that they make the program more difficult to read, to maintain, and to port to a new system. In particular, a port may require adding yet another alternative to a block of condition inclusions that is repeated in many places in the program.

Include files can be used to ameliorate this problem, as shown in Figure 2. This approach does improve readability, but it requires a separate small file for each pragma, and a potentially large number of inclusions of these files. Unless a translator is aggressive about caching and reusing included files, this can markedly increase compile time. For example, the compile time measured for files containing 100 replications of the examples in Figure 1 and Figure 3 differed by less than 3% for one compiler but by a factor of 3 for another.

Figure 2 *Pragmas in include files*

```
fast_call.h:
===========
    #if defined(Machine_A)
      /* Request fastest calling sequence for machine A */
    # pragma fast_call
    #elif defined(Machine_B)
      /* Request fastest calling sequence for machine B */
    # pragma vfunction
    #endif

independent.h:
==============
    #if defined(Machine_B)
      /* Vectorization hint (ignore vector dependencies) */
    # pragma ivdep
    #elif defined(Machine_C)
      /* Parallelization hint (iterations are independent) */
    # pragma independent
    #endif
```

Figure 3 *Included pragmas*

```
#include "fast_call.h"
void f(int n, double * a, double * b) {
    #include "independent.h"
    while(n-- > 0) {
        *a++ += *b++;
    }
}
```

But perhaps the most serious limitation of the directive syntax is that it cannot be included in the replacement-list for a macro. This means that if the function f in these examples is redefined as a macro, no directives can be specified in its body. There are cases where both a pragma and the conversion of the function containing it to a macro would significantly boost performance, and so it is unfortunate that the two cannot be combined.

2 Prior art for a solution

The Cray Standard C compiler supports an alternative syntax for pragmas which allows them to appear in a macro replacement-list, as shown in Figure 4.

Figure 4 *Pragmas in macros*

```
pragmas.h:
===========
    #if defined(Machine_A)
      /* Request fastest calling sequence for machine A */
    # define Fast_call \
              _Pragma("fast_call")
    #elif defined(Machine_B)
      /* Request fastest calling sequence for machine B */
    # define Fast_call \
              _Pragma("vfunction")
    #else
    # define Fast_call
    #endif

    #if defined(Machine_B)
      /* Vectorization hint (ignore vector dependencies) */
    # define Independent _Pragma("ivdep")
    #elif defined(Machine_C)
      /* Parallelization hint (iterations are independent) */
    # define Independent _Pragma("independent")
    #else
    # define Independent
    #endif
```

Here, the relevant macros for the various target systems are defined in a single header file as relatively general and mnemonic macros. This allows the previous examples to be expressed more clearly in Figure 5.

Figure 5 *Using the macros*

```
#include "pragmas.h"

Fast_call
void f(int n, double * a, double * b) {
    Independent
    while(n-- > 0) {
        *a++ += *b++;
    }
}
```

It also allows a pragma to be used in a macro version of the function.

Figure 6 *Pragmas in macroized function*

```
# define f(N, A, B) \
{   int n = (N), double * a = (A), double * b = (B); \
    Independent while(n-- > 0) { *a++ += *b++; } \
}
```

Note that in these examples, each type of `_Pragma` is hidden in a macro with a simple descriptive name, and that name is defined to be empty when the relevant type of pragma is not supported.

Even if `_Pragma(...)` appears explicitly at each point in the program where a pragma is needed, it still offers a portability advantage over the current directive. With the new syntax, all pragmas can be eliminated by simply defining a function-like macro named `_Pragma` with an empty replacement-list.

Finally, while the identifier `_Pragma` was appropriate for a vendor extension, the identifier `pragma`, from the user's name space, is proposed for the Standard.

3 Changes to the Standard

In section 5.1.1.2, paragraph 1, modify the first sentence of phase 4 to read:

> Preprocessing directives are executed, macro invocations are expanded, and pragma unary operator expressions are executed.

Add a new section 6.8.9, entitled "The pragma operator":

> In translation phase 4, each unary operator expression of the form:
>
> `pragma` (*string-literal*)
>
> is executed as follows. The *string-literal* is "de-stringized" 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. 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. The original four preprocessing tokens in the unary operator expression are replaced by the (possibly empty) implementation-defined sequence of preprocessing tokens that result from that execution.

Text for 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 pragma unary operator expressions and pragma directives should be treated consistently in whether they are preserved and in whether macro invocations within them are expanded.

4 Why use quotes?

Why not allow any sequence of preprocessing tokens within the parentheses, and eliminate the "destringize" step? This would make the specification a little simpler, but would probably make the implementation more difficult. The string-literal approach minimizes the interaction of the new feature with the rest of preprocessing, and allows it to be added to an existing implementation as a largely self-contained separate step. (In the Cray Standard C compiler, `_Pragma` is represented as a predefined, single-argument macro, which is given special treatment at just a few places in the preprocessing code. The conversion of the string-literal argument into preprocessing tokens is done in the same function that is used for the result of the `##` token gluing operator and also for a macro that is defined on the command line by the `-D` option.)

Another consideration is that C code is likely to be processed by tools other than the compiler, and it can be useful for `pragma` to be able to masquerade an an external function, as it can in the string-literal version.

Finally, if the string-literal version is available, the other syntax can be implemented by the programmer with a couple of macro definitions, as in Figure 7. (Note that a pragma that requires a comma-list not enclosed in quotes would be a problem for this approach, because the number of macro arguments for `PRAGMA` would be wrong.)

Figure 7 *Macro implementation of the alternative syntax*

```
#define EXPRAG(x) pragma(#x)
#define PRAGMA(x) EXPRAG(x)

int libfunc() { ... }

/* Direct the linker to define alternate entry points for libfunc. */
PRAGMA( duplicate libfunc as (lib_func,LIBFUNC) )
```

In view of these considerations, the string-literal approach seems preferable.

5 Conclusion

A small addition to the Standard allows pragmas to be included in a macro replacement-list. The benefits are that programs that make use of pragmas can be more readable and more easily ported, and can compile and execute more quickly on some systems.