Document Number: WG14 N531/X3J11 95-132

C9X Revision Proposal

Title: Assorted Preprocessor Extensions

Author: Frank Farance
Author Affiliation: Farance Inc.
Postal Address: 555 Main Street, New York, NY, 10044-0150, USA
E-mail Address: frank@farance.com
Telephone Number: +1 212 486 4700
Fax Number: +1 212 759 1605

Sponsor: X3J11
Date: 1995-12-22
Proposal Category:
__ Editorial change/non-normative contribution
__ Correction
X_ New feature
__ Addition to obsolescent feature list
__ Addition to Future Directions
__ Other (please specify) ______________________________
Area of Standard Affected:
__ Environment
__ Language
X_ Preprocessor
__ Library
  __ Macro/typedef/tag name
  __ Function
  __ Header
Prior Art:
Target Audience: C programmers
Related Documents (if any):
Proposal Attached: __ Yes X_ No, but what's your interest?
Abstract:

The following are various additions to the preprocessor. The improvements come from existing macro languages and the POSIX shell language. The purpose of these features is to be able to write more sophisticated preprocessor macros. Why? Compile-time (typed, but static) and run-time (typed and untyped, and dynamic) programming each have their advantages and disadvantages. The advantage of pre-processor programming is that it is untyped and static. Without these features, the programmer produces a less-than-optimal solution when using compile-time solutions (e.g., the operands must be typed -- this translates into larger code (multiple functions for different types), or limited function (everything is promoted to some type)) or run-time solutions (e.g., the typing is done at run-time, e.g., run-time typed identifiers and typing system).

This solution makes moderate extensions to the preprocessor in several areas: expanding a macro, evaluating a macro, preprocessor blocks, and preprocessor looping.

643

Since the preprocessor is complicated with many special cases, the first step should be deciding what kind of features we want. The second step is determining the precise semantics and standards wording.

EXPANDING A MACRO

This feature allows the programmer to completely expand a macro.

```
        #define a (b+c)+(d+e)
        #define b (x+y)
        #define z #expand(a)
```

This will define "z" as "((x+y) +c)+(d+e)".

GETTING A VALUE

This feature allows the programmer to evaluate an expression and produce its numeric value. For example:

```
        #define z 20
        #define y (z+10)
        #define x #value(y)
```

This statement calculates value of the expression "y", just as if it were used in a "#if" statement. This is useful for constant folding. In the above example, "x" is defined as "30", not "(z+10)". The "#value" directive is useful for creating temporary names:

```
        #define n 0

        /* ... */

        #define n #value(n+1)
        int temp_ ## #value(n) ;
```

Additionally, the preprocessor should support string comparison:

```
        #if IEEE_DOUBLE == "double"
```

This would be handled in the same way AWK determines whether to do a string or numeric comparison.

BLOCKS

Preprocessor blocks of code as single \`\`lines'', just like a block of C statements can act as a single statement:

```
        #define f(a,b,c) \
        #{
                #if defined(VAX)
                        vax_special_code(a,b,c);
                #else
                        regular_code(a,b,c);
                #endif
        #)
```

This is especially handy when embedding other preprocessor features (e.g., "#if") inside a definition.

LOOPING

This feature allows the programmer to write loops to generate code (e.g., initializing an array). For example:

```
        #define ARRAY_SIZE 10
        int array[ARRAY_SIZE] =
        {
        #for ( i = 0 ; i < ARRAY_SIZE ; i = #value(i+1) ) \
        #{
                [i] = i*i ,
        #}
        };
```

The following looping constructs are provided:

```
        #for ( start ; test ; increment ) body

        #while ( test ) body

        #do body
        #while ( test )
```

With looping control structures, "#break" and "#continue" are useful and intuitive:

```
        #break

        #continue

        #break N /* breaks N block levels */

        #continue N /* continues loop at N block levels */
```

SUPPORT FOR VARIABLE LENGTH ARGUMENTS

The programmer uses syntax similar to C prototype syntax to indicate that the macro takes a varying list of arguments:

```
        #define error_printf(format,...) /* 1 or more arguments */
        #define x(...) /* 0 or more arguments */
```

Within the definition, "#1" refers to argument 1, "#2" refers to argument 2, and so on. "#9" is argument 9 plus a comma-separated list of the remaining arguments. For example, in

```
        error_printf(a,b,c,d,e,f,g,h,i,j,k,l)
```

645

"#9" is "i,j,k,l". "#0" refers to the complete, comma-separated argument list. The "#shift" directive shifts all the argments left and drops argument 1. This allows for processing arbitrarily long argument lists. "#?" contains the number of arguments in the list.

```
        #define arg_print(...) \
        #{
                #define n 0
                #while ( #? > 0 ) \
                #{
                        printf("arg[%d]: %d\n",n,#1);
                        #shift
                #}
        #}
```