

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.10.3.5 Scope of macro definitions

| | | |
|---|--|------|
| <p>macro definition lasts until</p> <p>linkage scope of identifiers</p> <p>macro definition emulate block scope</p> | <p>A macro definition lasts (independent of block structure) until a corresponding #undef directive is encountered or (if none is encountered) until the end of the preprocessing translation unit.</p> <p>Commentary</p> <p>There is no requirement that macros with the same names in different translation units have the same replacement lists, or both be object-like or function-like (neither linkage or scope apply to macro names).</p> <p>Common Implementations</p> <p>Very few implementations write information on macro definitions out to the generated object file. Although some static analysis tools save this information for later cross translation unit consistency checking.</p> <p>Coding Guidelines</p> <p>The issue of having the definition of macro names exist over some kind of restricted scope is discussed elsewhere.</p> | 1974 |
| <p>macro definition no significance after</p> <p>preprocessing directives deleted</p> | <p>Macro definitions have no significance after translation phase 4.</p> <p>Commentary</p> <p>All preprocessing directives are deleted at the end of translation phase 4.</p> <p>C90</p> <p>This observation is new in C99.</p> <p>C++</p> <p>This observation is not made in the C++ document.</p> | 1975 |
| <p>#undef</p> <p>predefined macros not #defined</p> <p>#define/#undef stack</p> | <p>A preprocessing directive of the form</p> <p style="margin-left: 20px;"># undef identifier new-line</p> <p>causes the specified identifier no longer to be defined as a macro name.</p> <p>Commentary</p> <p>The relatively high percentage of macro definitions that do not include a replacement list (see Table ??) shows the degree to which the status of being defined as a macro name is sufficient information for developer use. The #undef directive provides additional control over which identifiers are defined as macro names.</p> <p>The standard specifies a few identifiers that cannot be #undefed.</p> <p>Common Implementations</p> <p>Some translators support a -U translator option that can be used to override any predefined macros (either defined using the -D option or internally generated by the translator). Some prestandard translators handled #define/#undef in a stack-like fashion.</p> <p>Coding Guidelines</p> <p>Discussion of the #undef directive, in guideline documents or between developers, is relatively rare. Whether this is because use of this directive rare, or simply innocuous is not known.</p> <p>Usage</p> <p>Approximate 5% of all #undef directives occur before a #include directive (based on the visible form of the .c files).</p> | 1976 |

Table 1976.1: Occurrence of various sequences of preprocessing directives (as a percentage of all such sequences) that follow a **#undef** and reference the same identifier (e.g., 2.7% of the first occurrence of **#undef** are followed by one or more **#defines** followed by one or more **#undefs**). **#define** represents one or more **#define** preprocessing directives. **#undef** represents one or more **#undef** preprocessing directives. **#if[n]def** represents two or more **#ifs** and **#ifndefs**, in any order. **#und-def** represents one or more pairs of **#undef #define** preprocessing directives. Based on the visible form of the .c files.

| Following Directive Sequences | % |
|--------------------------------------|------|
| | 53.0 |
| #ifdef | 20.4 |
| #define | 16.2 |
| others | 4.8 |
| #define #undef | 2.7 |
| #if(n)def | 1.5 |
| #define #undef-#define #undef | 1.4 |

1977 It is ignored if the specified identifier is not currently defined as a macro name.

Commentary

This behavior simplifies the process of using **#undef** by removing the need to check the status of the identifier (e.g., by using **#ifdef**) before evaluating the directive.

Coding Guidelines

The issue of redundant code is discussed elsewhere. Whether or not a **#undef** is redundant, or simply providing a fail safe backup is outside the scope of these guidelines.

redundant
code

1978 EXAMPLE 1 The simplest use of this facility is to define a “manifest constant”, as in

```
#define TABSIZE 100
int table[TABSIZE];
```

Commentary

Defining an object-like macro to have an integer constant replacement list is one of the most commonly given examples of the use of the preprocessor.

1979 EXAMPLE 2 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. Also it cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

Commentary

The issue of side effects in the evaluation of macro arguments is discussed elsewhere.

EXAMPLE
macro argu-
ment side effects

macro
arguments
separated by
comma

1980 EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)  f(x * (a))
#undef x
#define x      2
#define g      f
#define z      z[0]
#define h      g(~
```

EXAMPLE
reexamination

```

#define m(a)  a(w)
#define w    0,1
#define t(a)  a
#define p()  int
#define q(x)  x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
    (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };

```

results in

```

f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello" "" };

```

Commentary

Analyzing a part of the example, we get:

```

1  #define f(a)  f(x * (a))
2  #define x    2
3  #define z    z[0]
4
5  f(f(z));

```

the sequence of expansions are (preprocessing tokens bracketed by {} are no longer available for further replacement):

```

1  f(f(z))

```

replacing the argument z:

```

1  f(f({z}[]))

```

substituting and expanding the nested function-like macro:

```

1  f({f}(2 * ({z}[0])))

```

the argument has been fully expanded and can now be substituted into the replacement list:

```

1  f(2 * ({f}(2 * ({z}[0])))

```

and expanding the replacement list we get:

```

1  {f}(2 * ({f}(2 * ({z}[0])))

```

so the final sequence of preprocessing tokens is:

```

1  f(2 * (f(2 * (z[0])))

```

1981 EXAMPLE 4 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

EXAMPLE
and

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s " = %d, x" # t " = %s" \
                          x ## s, x ## t)

#define INCFILE(n) vers ## n
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW    "hello"
#define LOW        LOW " ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d" "abc", '\\4') // this goes away
      == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" " = %d, x" "2" " = %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" " ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

Commentary

The characters “(after macro replacement, before file access)” are commentary and are not generated by macro replacement.

1982 EXAMPLE 5 To illustrate the rules for placemaker preprocessing tokens, the sequence

EXAMPLE
placemaker

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
           10, 11, 12, };
```

Commentary

Although the order of evaluation of the ## operator is unspecified, in the above example all orders return the same result.

evaluation order

C90

This example is new in the C99 Standard and contains undefined behavior in C90.

C++

The C++ Standard specification is the same as that in the C90 Standard,

EXAMPLE
macro redefini-
tion

EXAMPLE 6 To demonstrate the redefinition rules, the following sequence is valid.

1983

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)  ( a )
#define FUNC_LIKE( a )(          /* note the white space */ \
                          a /* other stuff on this line
                          */ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE      (0) // different token sequence
#define OBJ_LIKE      (1 - 1) // different white space
#define FUNC_LIKE(b) ( a ) // different parameter usage
#define FUNC_LIKE(b) ( b ) // different parameter spelling
```

Commentary

The preprocessor is not required to have any knowledge of how subsequent phrases of translation treated sequences of preprocessing tokens (i.e., the fact that they become an integer constant expression that has the same value as the replacement list of another macro definition).

EXAMPLE
variable macro
arguments

EXAMPLE 7 Finally, to show the variable argument list macro facilities:

1984

```
#define debug(...)      fprintf(stderr, __VA_ARGS__)
#define showlist(...)   puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                          printf(__VA_ARGS__))

debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr, "Flag" );
fprintf(stderr, "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
    printf("x is %d but y is %d", x, y));
```

C90

Support for macros taking a variable number of arguments is new in C99.

C++

Support for macros taking a variable number of arguments is new in C99 and is not supported in C++.

References