Document Number: WG14 N419/X3J11 95-020

C9X Revision Proposal

Title: Addition of predefined identifier "__FUNC__".
Author: David R. Tribble
Author Affiliation: (Self)
Postal Address: 6004 Cave River Dr.
Plano, TX 75093-6951
USA
E-mail Address: drt@wcwcen.wf.com
drt@merlin.etsu.edu
Telephone Number: +1 214 9641720
+1 214 9608649 219 (day)
Fax Number: (None)
Sponsor: _____
Date: 1995-01-23
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
X_ Language
X_ Preprocessor
__ Library
__ Macro/typedef/tag name
__ Function
__ Header
__ Other (please specify) _____
Prior Art: I think this was implemented by Aztec C some years ago.
Target Audience: Programmers using debugging statements.
Related Documents (if any): (None)
Proposal Attached: X_ Yes __ No, but what's your interest?
Abstract: The use of the predefined identifier "__FUNC__" allows programmers to specify the name of the enclosing function in debugging statements.
It may also be used in the "assert()" macro.

========== Cover sheet ends here ==========

Proposal:

Notational Notes:
Font changes within source code examples are bracketed by troff-like notation, e.g., \fFfixed-width\fP, \fIitalics\fP, and \fBbold\fP; \fF causes a switch to fixed-width (courier) font, \fI chooses italic font, \fB chooses bold font, and \fP returns to the previous font. This is only used when it is confusing to use the recommended "courier", *italics*, and BOLD notations.

Problem Statement:

Programmers who use debugging statements often use the "__LINE__" and "__FILE__" predefined macro names ([6.8.8]). However, displaying the name of the function that contains a given debugging statement is not as easy.
Typically, programmers have to add hard-coded strings to their debugging statements (which are typically calls to "printf()").
This leads to clutter and wasted data space.
It also leads to confusion if function names are changed or function code is duplicated, and the programmer, in his haste, forgets to change the affected debugging statements.

Conceptual Model:
The addition of the predefined identifier "__FUNC__" would remedy this situation.

Syntax and Semantics:
The identifier "__FUNC__" is recognized by the translator as a predefined name with the implicit definition of:

\fF
```
static const char __FUNC__[\fIN\fP+1] =
                                    "\fIfunction_name\fP";
```
\fP

Its value is a null-terminated string constant containing the name of the current function (i.e., the function that contains the statement in which "__FUNC__" occurs).
Its size, *N*, is the length of the function's name.

For example:

\fF
```
#include <stdio.h>

int f(int a, int b)
{
    printf("%s(%d, %d)\n", __FUNC__, a, b);
    return a+b;
}
```
\fP

Whenever function "f" is called, its name and argument values are printed to the standard output.

A more typical use of "__FUNC__" is for debugging statements.

It can also be used by the "assert()" macro (defined in <assert.h> [7.2.1.1]), allowing the name of the enclosing function to be printed in addition to the "__FILE__" and "__LINE__" macro values.

Rationale:
The use of the predefined "__FUNC__" identifier allows the programmer to use debugging statements of a more informatory nature, specifically, to indicate the particular function of interest.
This is especially useful for fatal errors and ''should not occur'' conditions that produce warnings for the end user. (Well, not for the end user per se, but for technical support personnel rendering aid to an end user who encounters such a warning.)

Constraints:
Unlike the "__FILE__" identifier, the "__FUNC__" name cannot be interpreted by most compilers during the preprocessing phases of translation (phases 1 through 4 [5.1.1.2]), but must be dealt with during the syntactical parsing phase (phase 7 [5.1.1.2]), similar to the "sizeof" operator.
It would most likely be treated as a predefined variable name whose scope is limited to (i.e., local to) each function.

The "__FUNC__" identifier is defined only within function definitions (i.e., within the *compound-statement* comprising the body of a function [6.7.1]), and is undefined outside of function definitions.
For upward compatibility with C++, it may make more sense to allow it to be defined immediately after the function's opening "(" of its parameter list, up to the closing "}" of the function body, so that function parameters with default values may be assigned this value.

Implementation Issues:
The simplest way to implement "__FUNC__" is for the compiler to define a constant string containing the name of the function currently being parsed.
Each instance of "__FUNC__" within the body of the function is semantically equivalent to an instance of the string constant.

This scheme is simple, but another scheme that is not so wasteful of data space is for the compiler to allocate a single constant string variable for each function.
This eliminates redundant duplicate data for every reference to "__FUNC__".

A further optimization is to generate a data definition for "__FUNC__" for a given function only if it is actually referenced within that function.

One snag needs to be resolved, however.
Since "__FILE__", "__LINE__", and the other predefined macro identifiers ([6.8.8]) are defined in the preprocessing phase of translation, it is possible to test for them:
```
\fF
    #ifdef __FILE__
    ...
    #endif
\fP
```
It would be advantageous, or it would at least make the language more orthogonal, to provide the same capability for "__FUNC__".
That is, define "__FUNC__" so that the preprocessor treats it like a predefined macro name for the purposes of "#if" directives, but such that the preprocessor does not actually replace it with anything.
It would be as if this directive had been processed (using the rules of recursive macro definitions [6.8.3.4]) prior to the inclusion of any program source lines:
```
\fF
```

```
    #define __FUNC__ __FUNC__
 \fP
```

Subsetting:
If an implementor chooses not to implement this feature, the programmer could determine this using code like the following:

```
 \fF
    #ifdef __FUNC__
        printf("%s: %d\n", __FUNC__, __LINE__);
    #else
        printf("%s: %d\n", "subr", __LINE__);
    #endif
 \fP
```

========== End of Proposal ==========