The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.4.2.2 Predefined identifiers

Semantics

__func__

The identifier **__func__** shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration 810

```
static const char __func__[] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.[61)]

Commentary

Implicitly declaring **__func__** immediately after the opening brace in a function definition means that the first, developer-written declaration within that function can access it. Giving **__func__** static storage duration enables its address to be referred to outside the lifetime of the function that contains it (e.g., enabling a call history to be displayed at some later stage of program execution). This is not a storage overhead because space needs to be allocated for the string literal denoted by **__func__**. The **const** qualifier ensures that any attempts to modify the value cause undefined behavior. The identifier **__func__** has an array type, and is not a string literal, so the string concatenation that occurs in translation phase 6 is not applicable.

translation phase 6

This identifier is useful for providing execution trace information during program testing. Developers who make use of UCNs may need to ensure that the library they use supports the character output required by them:

```
#include <stdio.h>

void \u30CE(void)
{
printf ("Just entered %s\n", __func__);
}
```

identifier multibyte character in

The issue of wide characters in identifiers is discussed elsewhere.

Which function name is used when a function definition contains the **inline** function specifier? In:

```
#include <stdio.h>

inline void f(void)
{
printf("We are in %s\n", __func__);
}

int main(void)
{
f();
printf("We are in %s\n", __func__);
}
```

the name of the function `f` is output, even if that function is inlined into `main`.

C90

Support for the identifier **__func__** is new in C99.

C++

Support for the identifier **__func__** is new in C99 and is not available in the C++ Standard.

Common Implementations

A translator only needs to declare **__func__** if a reference to it occurs within a function. An obvious storage saving optimization is to delay any declaration until such time as it is known to be required. Another optimization is for the storage allocated for **__func__** to exactly overlay that allocated to the string literal. Allocating storage for a string literal and copying the characters to the separately allocated object it initializes is not necessary when that object is defined using the **const** qualifier. `gcc` also supports the built-in form **__FUNCTION__**.

Example

Debugging code in functions can provide useful information. But when there are lots of functions, the quantity of useless information can be overwhelming. Controlling which functions are to output debugging information by using conditional compilation requires that code be edited and the program rebuilt.

The names of functions can be used to dynamically control which functions are to output debugging information. This control not only reduces the amount of information output, but can also reduce execution time by orders of magnitude (output can be a resource-intense operation).

flookup.h

```
typedef struct f__rec {
                      char *func_name;
                      _Bool enabled;
                      struct f__rec *next;
                      } func__list;

extern _Bool func_lookup(func__list *, char *);

/*
 * Use the name of the function to control whether debugging is
 * switched on/off.  lookup is only called the first time this code
 * is executed, thereafter the value f___l->enabled can be used.
 */
#define D_func_trace(func_name, code)  {           \
                 static func__list * f___l = NULL; \
                 if (f___l ? f___l->enabled : lookup(&f___l, func_name)) \
                    {code}                                               \
                 }
```

flookup.c

```
#include <stdbool.h>

#include "flookup.h"

/*
 * A fixed list of functions and their debug mode.
 * We could be more clever and make this a list which
 * could be added to as a program executes.
 */
static struct {
              char *func_name;
              _Bool enabled;
              func__list *traces_seen;
              } lookup_table[] = {
                                  "abc", true, NULL,
                                  NULL, false, NULL
                                  };

_Bool func_lookup(func__list *f_list, char *f_name)
{
/*
 * Loop through lookup_table looking for a match against f_name.
 * If a match is found, add f_list to the traces_seen list and
 * return the value of enabled for that entry.
 */
}

void change_enabled_setting(char *f_name, _Bool new_enabled)
{
/*
 * Loop through lookup_table looking for a match against f_name.
 * If a match is found, loop over its traces_seen list setting
```

```
33    * the enabled flag to new_enabled.
34    *
35    * This function can switch on/off the debugging output from
36    * any registered function.
37    */
38   }
```

This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5. 811

Commentary

translation phase 5

Having the name appearing as if in translation phase 5 avoids any potential issues caused by macro names defined with the spelling of keywords or the name **__func__**. It also enables a translator to have an identifier name and type predefined internally, ready to be used when this reserved identifier is encountered. Translation phase 5 is also where characters get converted to their corresponding members in the execution character set, an essential requirement for spelling a function name. In many implementations the function name written to the object file, or program image, is different from the one appearing in the source. This translation phase 5 requirement ensures that it is not any modified name that is used.

program image

Example

```
1   #include <stdio.h>
2
3   #define __func__ __CNUF__
4   #define __CNUF__ "g"
5
6   void f(void)
7   {
8   /*
9    * The implicit declaration does not appear until after preprocessing.
10   * So there is no declaration 'static const char __func__[] = "f";'
11   * visible to the preprocessor (which would result in __func__ being
12   * mapped to __CNUF__ and "f" rather than "g" being output).
13   */
14  printf("Name of function is %s\n", __CNUF__);
15  }
```

EXAMPLE Consider the code fragment: 812

```
#include <stdio.h>
void myfunc(void)
{
        printf("s\n", __func__);
        /* ... */
}
```

Each time the function is called, it will print to the standard output stream:
myfunc

Commentary

This assumes that the standard output stream is not closed (in which case the behavior would be undefined).

Forward references: function definitions (6.9.1). 813

814 61) Since the name **__func__** is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name **__func__**, the behavior is undefined.

footnote 61

Commentary

The name is reserved because it begins with two underscores. The fact that the standard defines an interpretation for this name in the identifier name space in block scope does not give any license to the developer to use it in other name spaces or at file scope. This name is still reserved for use in other name spaces and scopes.

C90

Names beginning with two underscores were specified as reserved for any use by the C90 Standard. The following program is likely to behave differently when translated and executed by a C99 implementation.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  int __func__ = 1;
6
7  printf("d\n", __func__);
8  }
```

C++

Names beginning with __ are reserved for use by a C++ implementation. This leaves the way open for a C++ implementation to use this name for some purpose.

References

1. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.