C multi-line macro: do/while(0) vs scope block [duplicate]

Possible Duplicates:
What's the use of do while(0) when we define a macro?
Why are there sometimes meaningless do/while and if/else statements in C/C++ macros?
do { … } while (0) what is it good for?

I've seen some multi-line C macros that are wrapped inside a do/while(0) loop like:

```
#define FOO \
  do { \
    do_stuff_here \
    do_more_stuff \
  } while (0)
```

What are the benefits (if any) of writing the code that way as opposed to using a basic block:

```
#define FOO \
  { \
    do_stuff_here \
    do_more_stuff \
  }
```

c macros multiline

edited May 23 at 12:18
Community ♦
1 1

asked Jul 1 '09 at 4:06
krasnaya
1,119 3 11 16

marked as duplicate by sth, Adam Rosenfield, Emil H, Jonathan Leffler, Ialto Jul 1 '09 at 7:10

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

4 Duplicates: stackoverflow.com/questions/923822 and stackoverflow.com/questions/154136 and stackoverflow.com/questions/257418 – sth Jul 1 '09 at 4:19

Actually there is another way to get things right. ({...}) can do the same thing as do {} while(0). Ref: bruceblinn.com/linuxinfo/DoWhile.html – Nybble Aug 3 '16 at 19:51

1 Answer

http://bytes.com/groups/c/219859-do-while-0-macro-substitutions

Andrey Tarasevich:

The whole idea of using 'do/while' version is to make a macro which will expand into a regular statement, not into a compound statement. This is done in order to make the use of function-style macros uniform with the use of ordinary functions in all contexts.

Consider the following code sketch

```
if (<condition>)
  foo(a);
else
  bar(a);
```

where 'foo' and 'bar' are ordinary functions. Now imagine that you'd like to replace function 'foo' with a macro of the above nature

```
if (<condition>)
  CALL_FUNCS(a);
else
  bar(a);
```

Now, if your macro is defined in accordance with the second approach (just '{' and '}') the code will no longer compile, because the 'true' branch of 'if' is now represented by a compound statement. And when you put a ';' after this compound statement, you finished the whole 'if' statement, thus orphaning the 'else' branch (hence the compilation error).

One way to correct this problem is to remember not to put ';' after macro "invocations"

```
if (<condition>)
  CALL_FUNCS(a)
else
  bar(a);
```

This will compile and work as expected, but this is not uniform. The more elegant solution is to make sure that macro expand into a regular statement, not into a compound one. One way to achieve that is to define the macro as follows

```
#define CALL_FUNCS(x) \
do { \
  func1(x); \
  func2(x); \
  func3(x); \
} while (0)
```

Now this code

```
if (<condition>)
  CALL_FUNCS(a);
else
  bar(a);
```

will compile without any problems.

However, note the small but important difference between my definition of `CALL_FUNCS` and the first version in your message. I didn't put a `;` after `} while (0)`. Putting a `;` at the end of that definition would immediately defeat the entire point of using 'do/while' and make that macro pretty much equivalent to the compound-statement version.

I don't know why the author of the code you quoted in your original message put this `;` after `while (0)`. In this form both variants are equivalent. The whole idea behind using 'do/while' version is not to include this final `;` into the macro (for the reasons that I explained above).

edited Aug 26 '14 at 21:52
Robert Harvey ♦
136k 30 242 372

answered Jul 1 '09 at 4:11
caskey
8,394 2 20 25

31 The original post was made by me in `comp.lang.c`. `bytes.com` apparently "appropriates" the content of `comp.lang.c` without making any references to the source. – AnT Oct 23 '09 at 6:30