

Why use apparently meaningless do-while and if-else statements in macros?

Asked 16 years, 2 months ago Modified 3 days ago Viewed 121k times



In many C/C++ macros I'm seeing the code of the macro wrapped in what seems like a meaningless `do while` loop. Here are examples.

954



```
#define F00(X) do { f(X); g(X); } while (0)
#define F00(X) if (1) { f(X); g(X); } else
```



I can't see what the `do while` is doing. Why not just write this without it?



```
#define F00(X) f(X); g(X)
```

c++

c

c-preprocessor

c++-faq

Share

Improve this question

Follow

edited Dec 10 at 10:21



Donald Duck

8,831 ● 23 ● 79 ● 101

asked Sep 30, 2008 at 17:36



jfm3

37.7k ● 10 ● 34 ● 35

4 For the example with the else, I would add an expression of `void` type at the end... like `((void)0)`. – [Phil1970](#) Mar 8, 2018 at 17:59

4 Reminder that the `do while` construct isn't compatible with return statements, so the `if (1) { ... } else ((void)0)` construct has more compatible usages in Standard C. And in GNU C, you'll prefer the construct described in my answer. – [Cœur](#) Dec 12, 2018 at 11:50

9 Answers

Sorted by: Highest score (default)



The `do ... while` and `if ... else` are there to make it so that a semicolon after your macro always means the same thing. Let's say you had something like your second macro.

1000



```
#define BAR(X) f(x); g(x)
```



Now if you were to use `BAR(X);` in an `if ... else` statement, where the bodies of the if statement were not wrapped in curly brackets, you'd get a bad surprise.



```
if (corge)
    BAR(corge);
else
    gralt();
```

The above code would expand into

```
if (corge)
    f(corge); g(corge);
else
    gralt();
```

which is syntactically incorrect, as the else is no longer associated with the if. It doesn't help to wrap things in curly braces within the macro, because a semicolon after the braces is syntactically incorrect.

```
if (corge)
    {f(corge); g(corge);};
else
    gralt();
```

There are two ways of fixing the problem. The first is to use a comma to sequence statements within the macro without robbing it of its ability to act like an expression.

```
#define BAR(X) f(X), g(X)
```

The above version of bar `BAR` expands the above code into what follows, which is syntactically correct.

```
if (corge)
    f(corge), g(corge);
else
    gralt();
```

This doesn't work if instead of `f(x)` you have a more complicated body of code that needs to go in its own block, say for example to declare local variables. In the most general case the solution is to use something like `do ... while` to cause the macro to be a single statement that takes a semicolon without confusion.

```
#define BAR(X) do { \
    int i = f(X); \
    if (i > 4) g(i); \
} while (0)
```

You don't have to use `do ... while`, you could cook up something with `if ... else` as well, although when `if ... else` expands inside of an `if ... else` it leads to a "[dangling else](#)", which could make an existing dangling else problem even harder to find, as in the following code.

```
if (corge)
    if (1) { f(corge); g(corge); } else;
else
    gralt();
```

The point is to use up the semicolon in contexts where a dangling semicolon is erroneous. Of course, it could (and probably should) be argued at this point that it would be better to declare `BAR` as an actual function, not a macro.

In summary, the `do ... while` is there to work around the shortcomings of the C preprocessor. When those C style guides tell you to lay off the C preprocessor, this is the kind of thing they're worried about.

Share

Improve this answer

Follow

edited Dec 10 at 10:19



Donald Duck

8,831 ● 23 ● 79 ● 101

answered Sep 30, 2008 at 17:36



jfm3

37.7k ● 10 ● 34 ● 35

-
- 43 Isn't this a strong argument to always use braces in `if`, `while` and `for` statements? If you make a point of always doing this (as is required for MISRA-C, for example), the problem described above goes away. – [Steve Melnikoff](#) Apr 2, 2009 at 22:51
-
- 26 The comma example should be `#define BAR(X) (f(X), g(X))` otherwise operator precedence may mess up the semantics. – [Stewart](#) May 19, 2011 at 12:58
-
- 27 @DawidFerenczy: although both you and me-from-four-and-a-half-years-ago make a good point, we have to live in the real world. Unless we can guarantee that all the `if` statements, etc, in our code use braces, then wrapping macros like this is a simple way of avoiding problems. – [Steve Melnikoff](#) Nov 20, 2013 at 17:16
-
- 23 Note: the `if(1) {...} else void(0)` form is safer than the `do {...} while(0)` for macros whose parameters are code that is included in the macro expansion, because it doesn't alter the behavior of the `break` or `continue` keywords. For example: `for (int i = 0; i < max; ++i) { MYMACRO(SomeFunc(i)==true, {break;}) }` causes unexpected behavior when `MYMACRO` is defined as `#define MYMACRO(X, CODE) do { if (X) { cout << #X << endl; {CODE}; } } while (0)` because the `break` affects the macro's `while` loop rather than the `for` loop at the macro call site. – [Chris Kline](#) Jun 22, 2015 at 16:23 ✎
-
- 10 @ace `void(0)` was a typo, I meant `(void)0`. And I believe this *does* solve the "dangling else" problem: notice there's no semicolon after the `(void)0`. A dangling else in that case (e.g. `if (cond) if (1) foo() else (void)0 else { /* dangling else body */ }`) triggers a compilation error. Here's a [live example demonstrating it](#) – [Chris Kline](#) Jul 27, 2015 at 11:07 ✎
-



188



Macros are copy/pasted pieces of text the pre-processor will put in the genuine code; the macro's author hopes the replacement will produce valid code.

There are three good "tips" to succeed in that:

Help the macro behave like genuine code

Normal code is usually ended by a semi-colon. Should the user view code not needing one...

```
doSomething(1) ;  
DO_SOMETHING_ELSE(2) // <== Hey? What's this?  
doSomethingElseAgain(3) ;
```

This means the user expects the compiler to produce an error if the semi-colon is absent.

But the real real good reason is that at some time, the macro's author will perhaps need to replace the macro with a genuine function (perhaps inlined). So the macro should **really** behave like one.

So we should have a macro needing semi-colon.

Produce a valid code

As shown in jfm3's answer, sometimes the macro contains more than one instruction. And if the macro is used inside a if statement, this will be problematic:

```
if(bIsOk)  
    MY_MACRO(42) ;
```

This macro could be expanded as:

```
#define MY_MACRO(x) f(x) ; g(x)  
  
if(bIsOk)  
    f(42) ; g(42) ; // was MY_MACRO(42) ;
```

The `g` function will be executed regardless of the value of `bIsOk`.

This means that we must have to add a scope to the macro:

```
#define MY_MACRO(x) { f(x) ; g(x) ; }
```

```
if(bIsOk)
    { f(42) ; g(42) ; } ; // was MY_MACRO(42) ;
```

Produce a valid code 2

If the macro is something like:

```
#define MY_MACRO(x) int i = x + 1 ; f(i) ;
```

We could have another problem in the following code:

```
void doSomething()
{
    int i = 25 ;
    MY_MACRO(32) ;
}
```

Because it would expand as:

```
void doSomething()
{
    int i = 25 ;
    int i = 32 + 1 ; f(i) ; ; // was MY_MACRO(32) ;
}
```

This code won't compile, of course. So, again, the solution is using a scope:

```
#define MY_MACRO(x) { int i = x + 1 ; f(i) ; }

void doSomething()
{
    int i = 25 ;
    { int i = 32 + 1 ; f(i) ; } ; // was MY_MACRO(32) ;
}
```

The code behaves correctly again.

Combining semi-colon + scope effects?

There is one C/C++ idiom that produces this effect: The do/while loop:

```
do
{
    // code
}
while(false) ;
```

The do/while can create a scope, thus encapsulating the macro's code, and needs a semi-colon in the end, thus expanding into code needing one.

The bonus?

The C++ compiler will optimize away the do/while loop, as the fact its post-condition is false is known at compile time. This means that a macro like:

```
#define MY_MACRO(x) \
do \
{ \
    const int i = x + 1 ; \
    f(i) ; g(i) ; \
} \
while(false)

void doSomething(bool bIsOk)
{
    int i = 25 ;

    if(bIsOk)
        MY_MACRO(42) ;

    // Etc.
}
```

will expand correctly as

```
void doSomething(bool bIsOk)
{
    int i = 25 ;

    if(bIsOk)
    do
    {
        const int i = 42 + 1 ; // was MY_MACRO(42) ;
        f(i) ; g(i) ;
    }
    while(false) ;

    // Etc.
}
```

and is then compiled and optimized away as

```
void doSomething(bool bIsOk)
{
    int i = 25 ;

    if(bIsOk)
    {
        f(43) ; g(43) ;
    }
}
```

```
// Etc.  
}
```

Share

edited Dec 10 at 10:28

answered Sep 30, 2008 at 18:12

Improve this answer



Donald Duck

8,831 ● 23 ● 79 ● 101



paercebal

83.2k ● 38 ● 134 ● 160

Follow

7 Note that changing macros to inline function alters some standard predefined macros, e.g. the following code shows a change in **FUNCTION** and **LINE**: `#include <stdio.h> #define Fmacro() printf("%s %d\n", FUNCTION, LINE) inline void Finline() { printf("%s %d\n", FUNCTION, LINE); } int main() { Fmacro(); Finline(); return 0; }` (bold terms should be enclosed by double underscores — bad formatter!) – [Gnubie](#) Aug 23, 2012 at 10:52 ✎

8 There are a number of minor but not completely inconsequential issues with this answer. For instance: `void doSomething() { int i = 25 ; { int i = x + 1 ; f(i) ; } ; // was MY_MACRO(32) ; }` is not the correct expansion; the `x` in the expansion should be 32. A more complex issue is what is the expansion of `MY_MACRO(i+7)`. And another is the expansion of `MY_MACRO(0x07 << 6)`. There's a lot that's good, but there are some undotted i's and uncrossed t's. – [Jonathan Leffler](#) Aug 26, 2013 at 4:17

@Gnubie: I case you're still here and you haven't figured this out by now: you can escape asterisks and underscores in comments with backslashes, so if you type `__LINE__` it renders as `__LINE__`. IMHO, it's better just to use code formatting for code; for example, `__LINE__` (which doesn't require any special handling).

P.S. I don't know whether this was true in 2012; they've made quite a few improvements to the engine since then. – [Scott - Слава Україні](#) Aug 22, 2017 at 23:07 ✎

1 Appreciating that my comment is six years late, but most C compilers do not actually inline `inline` functions (as permitted by the standard) – [Andrew](#) Jan 3, 2019 at 6:01

Thanks! The most upvoted answer covered that you need a `while(0)` to behave correctly in unscoped `if` statements. But what they didn't cover (and you did) is why it isn't simply enough to put the macro body in a scope, like `#define F00(x) { bar(x); }`. It seems the real reason that you need `while(0)` is to require a semi-colon after the statement. – [Snackoverflow](#) Jul 31, 2023 at 14:04 ✎



58

@jfm3 - You have a nice answer to the question. You might also want to add that the macro idiom also prevents the possibly more dangerous (because there's no error) unintended behavior with simple 'if' statements:



```
#define F00(x) f(x); g(x)  
  
if (test) F00( baz);
```

expands to:

```
if (test) f(baz); g(baz);
```

which is syntactically correct so there's no compiler error, but has the probably unintended consequence that `g()` will always be called.

Share

edited Dec 10 at 10:29

community wiki

Improve this answer

2 revs, 2 users 74%

Michael Burr

Follow



25

The above answers explain the meaning of these constructs, but there is a significant difference between the two that was not mentioned. In fact, there is a reason to prefer the `do ... while` to the `if ... else` construct.



The problem of the `if ... else` construct is that it does not *force* you to put the semicolon. Like in this code:



```
F00(1)
printf("abc");
```

Although we left out the semicolon (by mistake), the code will expand to

```
if (1) { f(X); g(X); } else
printf("abc");
```

and will silently compile (although some compilers may issue a warning for unreachable code). But the `printf` statement will never be executed.

`do ... while` construct does not have such problem, since the only valid token after the `while(0)` is a semicolon.

Share Improve this answer Follow

answered Aug 3, 2012 at 15:21



Yakov Galka

72.3k ● 16 ● 145 ● 222


3 @RichardHansen: Still not as good, because from the look at the macro invocation you don't know whether it expands to a statement or to an expression. If someone assumes the later, she may write `F00(1), x++;` which will again give us a false positive. Just use `do ... while` and that's it. – Yakov Galka Aug 3, 2012 at 17:35

1 Documenting the macro to avoid the misunderstanding should suffice. I do agree that `do ... while (0)` is preferable, but it has one downside: A `break` or `continue` will control the `do ... while (0)` loop, not the loop containing the macro invocation. So the `if` trick still has value. – Richard Hansen Aug 3, 2012 at 21:10

3 I don't see where you could put a `break` or a `continue` that would be seen as inside your macros `do {...} while(0)` pseudo-loop. Even in the macro parameter it would make a syntax error. – Patrick Schlüter Dec 21, 2012 at 16:02

- 7 Another reason to use `do { ... } while(0)` instead of `if whatever` construct, is the idiomatic nature of it. The `do { ... } while(0)` construct is widespread, well known and used a lot by many programmers. Its rationale and documentation is readily known. Not so for the `if` construct. It takes therefore less effort to grok when doing code review.

– [Patrick Schlüter](#) Dec 21, 2012 at 16:05

- 2 @tristopia: I've seen people write macros that take blocks of code as arguments (which I don't necessarily recommend). For example: `#define CHECK(call, onerr) if (0 != (call)) { onerr } else (void)0`. It could be used like `CHECK(system("foo"), break;;)`, where the `break;` is intended to refer to the loop enclosing the `CHECK()` invocation. – [Richard Hansen](#) Jul 18, 2013 at 17:18 

Explanation

24

`do {} while (0)` and `if (1) {} else` are to make sure that the macro is expanded to only 1 statement. Otherwise:

```
if (something)
    FOO(X);
```

would expand to:

```
if (something)
    f(X); g(X);
```

And `g(X)` would be executed outside the `if` control statement. This is avoided when using `do {} while (0)` and `if (1) {} else`.

Better alternative

With a GNU [statement expression](#) (not a part of standard C), you have a better way than `do {} while (0)` and `if (1) {} else` to solve this, by simply using `({})`:

```
#define FOO(X) ({f(X); g(X);})
```

And this syntax is compatible with return values (note that `do {} while (0)` isn't), as in:

```
return FOO("X");
```

Share

Improve this answer

Follow

edited Dec 17 at 11:49



Toby Speight

30.6k ● 49 ● 72 ● 113

answered Mar 23, 2014 at 12:04



Cœur

38.6k ● 26 ● 202 ● 276

- 5 the use of block-clamping `{ }` in the macro would be sufficient for bundling the macro code so that all is executed for the same if-condition path. the do-while around is used for enforcing a semicolon at places the macro gets used. thus the macro is enforced behaving more function alike. this includes the requirement for the trailing semicolon when used. – [Alexander Stohr](#) Dec 21, 2018 at 14:38 ✎



17



While it is expected that compilers optimize away the `do { ... } while(false);` loops, there is another solution which would not require that construct. The solution is to use the comma operator:

```
#define F00(X) (f(X),g(X))
```

or even more exotically:

```
#define F00(X) g((f(X),(X)))
```

While this will work well with separate instructions, it will not work with cases where variables are constructed and used as part of the `#define` :

```
#define F00(X) (int s=5,f((X)+s),g((X)+s))
```

With this one would be forced to use the do/while construct.

Share

Improve this answer

Follow

edited Dec 10 at 14:20



Donald Duck

8,831 ● 23 ● 79 ● 101

answered Oct 10, 2009 at 8:27



Marius

3,461 ● 1 ● 33 ● 36

thanks, since the comma operator does not guarantee execution order, this nesting is a way to enforce that. – [Marius](#) Jun 5, 2011 at 13:39

- 17 @Marius: False; the comma operator is a sequence point and thereby **does** guarantee execution order. I suspect you confused it with the comma in function argument lists. – [R.. GitHub STOP HELPING ICE](#) Apr 12, 2012 at 5:02

Just wanted to add that compilers are forced to preserve the program observable behavior so optimizing the do/while away isn't much of a big deal (assuming the compiler optimizations are correct). – [Marco A.](#) Mar 31, 2015 at 9:42

@MarcoA. while you are correct, I have found in the past that compiler optimization, while exactly preserving the function of code, but by changing around lines which would seem to do

nothing in the singular context, will break multithreaded algorithms. Case in point
`Peterson's Algorithm` . – [Marius](#) Jul 25, 2016 at 12:29

This will also not work for all kinds of constructs, although C, with the ternary operator and this, is rather expressive. – [mirabilos](#) Dec 21, 2016 at 11:45



12



Jens Gustedt's [P99 preprocessor library](#) (yes, the fact that such a thing exists blew my mind too!) improves on the `if(1) { ... } else` construct in a small but significant way by defining the following:

```
#define P99_NOP ((void)0)
#define P99_PREFER(...) if (1) { __VA_ARGS__ } else
#define P99_BLOCK(...) P99_PREFER(__VA_ARGS__) P99_NOP
```

The rationale for this is that, unlike the `do { ... } while(0)` construct, `break` and `continue` still work inside the given block, but the `((void)0)` creates a syntax error if the semicolon is omitted after the macro call, which would otherwise skip the next block. (There isn't actually a "dangling else" problem here, since the `else` binds to the nearest `if`, which is the one in the macro.)

If you are interested in the sorts of things that can be done more-or-less safely with the C preprocessor, check out that library.

Share Improve this answer Follow

answered Nov 19, 2014 at 16:19



[Isaac Schwabacher](#)

121 ● 1 ● 2

While very clever, this causes one to be bombarded with compiler warnings on potential dangling else. – [Segmented](#) Mar 26, 2015 at 19:20

- 2 You usually use macros to create a contained environment, that is, you never use a `break` (or `continue`) inside a macro to control a loop that started/ended outside, that's just bad style and hides potential exit points. – [mirabilos](#) Dec 21, 2016 at 11:51

There's also a preprocessor library in Boost. What is mind-blowing about it? – [Rene](#) Jun 1, 2017 at 8:45

The risk with `else ((void)0)` is that someone might be writing `YOUR_MACRO(), f();` and it will be syntactically valid, but never call `f()`. With `do while` it's a syntax error. – [melpomene](#) Jun 7, 2019 at 20:42

@melpomene so what about `else do; while (0) ?` – [Carl Lei](#) Aug 15, 2019 at 2:29



For some reasons I can't comment on the first answer...

9

Some of you showed macros with local variables, but nobody mentioned that you can't just use any name in a macro! It will bite the user some day! Why? Because the input arguments are substituted into your macro template. And in your macro examples you've use the probably most commonly used variable name `i`.

For example when the following macro

```
#define FOO(X) do { int i; for (i = 0; i < (X); ++i) do_something(i); } while (0)
```

is used in the following function

```
void some_func(void) {
    int i;
    for (i = 0; i < 10; ++i)
        FOO(i);
}
```


the macro will not use the intended variable `i`, that is declared at the beginning of `some_func`, but the local variable, that is declared in the `do ... while` loop of the macro.

Thus, never use common variable names in a macro!

Share Improve this answer Follow

answered Dec 21, 2011 at 18:42

 [Mike Meyer](#)
99 ● 1 ● 1

-
- 9 @Blaisorblade: Actually that's incorrect and illegal C; leading underscores are reserved for use by the implementation. The reason you've seen this "usual pattern" is from reading system headers ("the implementation") which must restrict themselves to this reserved namespace. For applications/libraries, you should choose your own obscure, unlikely-to-collide names without underscores, e.g. `mylib_internal__i` or similar.
– [R.. GitHub STOP HELPING ICE](#) Apr 12, 2012 at 5:05
-
- 2 @R.. You're right - I've actually read this in an "application", the Linux kernel, but it's an exception anyway since it uses no standard library (technically, a "freestanding" C implementation instead of a "hosted" one). – [Blaisorblade](#) Apr 12, 2012 at 16:23
-
- 4 @R.. this is not quite correct: leading underscores *followed by a capital or second underscore* are reserved for the implementation in all contexts. Leading underscores followed by something else are not reserved in local scope. – [Alex Celeste](#) May 3, 2014 at 3:15
-
- 1 @Leushenko: Yes, but the distinction is sufficiently subtle that I find it best to tell people just not to use such names at all. The people who understand the subtlety presumably already know that I'm glossing over the details. :-)
– [R.. GitHub STOP HELPING ICE](#) May 3, 2014 at 3:16 
-
- 1 While correct, this is not an answer to the question. By site rules, please make this into a comment; if you're unable to, gather karma from good answers (that actually are answers)

first; yes, first steps can be hard, but on SO there's just so much it's possible. – [mirabilos](#) Dec 21, 2016 at 11:47



1



I don't think it was mentioned so consider this

```
while(i<100)
    FOO(i++);
```

would be translated into

```
while(i<100)
    do { f(i++); g(i++); } while (0)
```

notice how `i++` is evaluated twice by the macro. This can lead to some interesting errors.

Share Improve this answer Follow

answered Oct 18, 2008 at 22:04



[John Nilsson](#)

17.3k ● 8 ● 35 ● 42

20 This has nothing to do with the do ... while(0) construct. – [Trent](#) Nov 18, 2008 at 19:54

2 True. But relevant to the topic of macros vs. functions and how to write a macro that behaves as a function... – [John Nilsson](#) Nov 26, 2008 at 19:21

3 Similarly to the above one, this is not an answer but a comment. On topic: that's why you use stuff only once: `do { int macroname_i = (i); f(macroname_i); g(macroname_i); } while (/* CONSTCOND */ 0)` – [mirabilos](#) Dec 21, 2016 at 11:49



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.