globals, re-entrant code and thread safety

I'm trying to decide whether to implement certain operations as macros or as functions.

Say, just as an example, I have the following code in a header file:

```
extern int x_GLOB;
extern int y_GLOB;

#define min(x,y) ((x_GLOB = (x)) < (y_GLOB = (y))? x_GLOB : y_GLOB)
```

the intent is to have each single argument computed only once (`min(x++,y++)` would not cause any problem here).

The question is: do the two global variables pose any issue in terms of code re-entrancy or thread safeness?

I would say no but I'm not sure.

And what about:

```
#define min(x,y) ((x_GLOB = (x)), \
                  (y_GLOB = (y)), \
                  ((x_GLOB < y_GLOB) ? x_GLOB : y_GLOB)
```

would it be a differnt case?

Please note that the question is not "in general" but is related to the fact that those global variables are used just within that macro and for the evaluation of a single expression.

Anyway, looking at the answers below, I think I can summarize as follows:

- It's not thread safe as nothing guarantees that a thread is suspended "in the middle" of the evaluation of an expression (as I hoped instead)
- The "state" those globals represent is, at least, the internal state of the "min" operation and preserving that state would, again at least, require to impose restrictions on how the function can be called (e.g. avoid min(min(1,2),min(3,1)).

I don't think using non-portable constructs is a good idea either so I guess the only option is to stay on the safe side an implement those cases as regular functions.

c macros

edited Jan 10 '10 at 17:31

asked Jan 10 '10 at 13:54

Remo.D
10.2k 3 25 50

How will min(min(a, b), min(c, d)) evaluate? – Lasse V. Karlsen Jan 10 '10 at 15:54

Point taken! Using the globals would require posing restriction on how this "function" should be used. THanks – Remo.D Jan 10 '10 at 16:07

6 Answers

Personally I don't think this code is safe at all, even if we don't consider multithreaded scenarios.

Take the following evaluation:

```
int a = min(min(1, 2), min(3, 4));
```

How will this expand properly and evaluate?

1. Evaluate min(1, 2) and assign value to x_GLOB (outer macro):
   - x_GLOB = 1
   - y_GLOB = 2
   - evaluate min, result is 1, assign to x_GLOB (for outer macro)
2. Evaluate min(3, 4) and assign value to y_GLOB (outer macro):
   - x_GLOB = 3 (uh-oh, 1 from the first expansion is now clobbered)
   - y_GLOB = 4
   - evaluate min, result is 3, assign to y_GLOB (for outer macro)
3. Evaluate min(a, b) where a is min(1, 2) and b is min(3, 4)
   - evaluate min of x_GLOB (which is 3) and y_GLOB (which is 3)
   - result of total evaluation is 3

Or am I missing something here?

answered Jan 10 '10 at 16:00

Lasse V. Karlsen
165k 50 346 533

Silently modifying global variables inside a macro like `min` is a very bad idea. You're much better off

- accepting that `min` can evaluate its argument multiple times (in which case I'd call it `MIN` to make it look less like a regular C function), or
- write a regular or inline function for `min`, or
- use gcc extensions to make the macro safe.

If you do any of these things, you eliminate the global variables entirely and hence any questions about reentracy and multi-threading.

Option 1:

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

Option 2:

```
int min(int x, int y) { return x < y ? x : y; }
```

Of course, this has the problem that you can't use this version of `min` for different argument types.

Option 3:

```
#define min(x, y) ({ \
    typeof(x) x__ = (x); \
    typeof(y) y__ = (y); \
    x__ < y__ ? x__ : y__; \
})
```

edited Jan 10 '10 at 14:16

answered Jan 10 '10 at 14:10

Dale Hagglund
7,892 2 13 30

+1; starting variable names with double underscores should be avoided, though... – Christoph Jan 10 '10 at 14:13

I always forget that that standard has usurped that entire subset of the identifier space. Even when I know that, I can barely make myself do something different. However, I'll fix it up here so as to set a good example. – Dale Hagglund Jan 10 '10 at 14:15

Probably worth noting that 'typeof' (used in option 3) is GCC only. – martinr Jan 10 '10 at 14:18

Names containing double underscores are also reserved. – anon Jan 10 '10 at 14:19

Really? In that case I give up. – Dale Hagglund Jan 10 '10 at 14:19

The globals are not thread-safe. You can make them so by using thread-local storage.

Personally, I'd use an inline function instead of a macro to avoid the problem altogether!

answered Jan 10 '10 at 14:06

Christoph
72.3k 19 97 149

Agreed. Macros best avoided and only good if you're using a compiler that does not honour inlining or is bad at inlining. – martinr Jan 10 '10 at 14:08

Thread-local storage is expensive; may be beter off using temporary local variables. The names of the locals to use could be passed to the macro. – martinr Jan 10 '10 at 14:13

There are two parts to this question, Reentrancy and thread safety. Reentrancy has a list of rules (see wikipedia link below) and thread safety requires either atomicity (supported at the hardware level) or OS-level syncrhonization. Thread-local storage + inline functions is one way to provide both, but not necessarily the best or fastest combination for all situations – Sam Post Jan 11 '10 at 7:44

Generally speaking, anytime you use global variables, your subroutine (or macro) is not re-entrant.

Anytime two threads access the same data or shared resource, you have to use synchronization primitives (mutexes, semaphores, etc) to coordinate access to the "critical section".

See Wikipedia reentrant page

answered Jan 10 '10 at 14:08

Sam Post
1,892 2 8 13

It's not thread-safe. To demonstrate this, suppose that thread A calls min(1,2) and threadB calls min(3,4). Suppose that thread A is running first, and is interrupted by the scheduler right at the question mark of the macro, because its time slice has expired.

Then x_GLOB is 1, and y_GLOB is 2.

Now suppose threadB runs for a while, and completes the contents of the macro:

x_GLOB is 3, y_GLOB is 4.

Now suppose threadA resumes. It will return x_GLOB (3) instead of the right answer (1).

Obviously I've simplified things a bit - being interrupted "at the question mark" is pretty handwavy, and threadA doesn't necessarily return 3, on some compilers with some optimisation levels it might keep the values in registers and not read back from x_GLOB at all. So the code emitted might happen to work with that compiler and options. But hopefully my example makes the point - you can't be sure.

I recommend you write a static inline function. If you're trying to be very portable, do it like this:

```
#define STATIC_INLINE static

STATIC_INLINE int min_func(int x, int y) { return x < y ? x : y; }

#define min(a,b) min_func((a),(b))
```

Then if you have a performance problem on some particular platform, and it turns out to be because that compiler fails to inline it, you can worry about whether on that compiler you need to define STATIC_INLINE differently (to `static inline` in C99, or `static __inline` for Microsoft, or whatever), or perhaps even implement the `min` macro differently. That level of optimisation ("does it get inlined?") isn't something you can do much about in portable code.

edited Jan 10 '10 at 16:10

answered Jan 10 '10 at 15:42

Steve Jessop
175k 15 235 488

That code is not safe for multithreading.

Just avoid macros, unless indispensable ("Efficient C++" books have examples of those cases).

answered Jan 10 '10 at 14:09

Ariel
3,103 1 23 44