Are recursive functions re-entrant

I have seen many recursive functions(mostly used in computing some mathematical operations e.g. factorial, sum of the digits in a number, etc...) which involve use of a static variable which holds the result of the each recursive call/operation, and uses it for the subsequent calls.

So does that make recursive functions non-rentrant and not thread-safe.

Are there other use-cases of recursive functions which does not need static variables?

recursion reentrancy

edited Jan 31 '10 at 11:50

Eimantas
31k 8 83 125

asked Jan 31 '10 at 11:31

goldenmean
5,563 19 81 153

7 Answers

> Are there other use-cases of recursive functions which does not need static variables.?

Of course. In fact, static variables in recursive functions should be the *exception*, not the rule:

> I have seen many recursive functions(mostly used in computing some mathematical operations e.g. factorial, sum of the digits in a number, etc...) which involve use of a static variable which holds the result of the each recursive call/operation, and uses it for the subsequent calls.

Those were quite frankly bad implementations. Static variables are absolutely not needed here. They probably served as accumulators; this can be done better by passing the accumulator around as an extra argument.

answered Jan 31 '10 at 11:39

Konrad Rudolph
250k 56 531 792

The two are different concepts. One does not imply the other, or vice versa.

For instance, is this a recursive function (hypothetical language)?

```
global sum = 0

proc accumulate(treeNode)
    sum += treeNode.Value
    if treeNode.Left then accumulate(treeNode.Left)
    if treeNode.Right then accumulate(treeNode.Right)
end
```

Obviously it is a recursive function, but it is not reentrant, due to the use of the global variable.
By "global" here, at the very least I mean "not local to the function".

However, this is a bad example, since it is very easy to make it not rely on the global variable at all, by simply returning the sum:

```
func accumulate(treeNode)
    sum = treeNode.Value
    if treeNode.Left then sum += accumulate(treeNode.Left)
    if treeNode.Right then sum += accumulate(treeNode.Right)
    return sum
end
```

There is nothing inherent in the concept of a recursive function that makes it non-threadsafe or reentrant, or the opposite, it all depends on what you actually write in the function in question.

answered Jan 31 '10 at 11:36


Lasse V. Karlsen
165k 50 346 533

Recursive functions where the state is passed via reference or higher data structure as an argument are reentrant.

answered Jan 31 '10 at 11:36


Ignacio Vazquez-Abrams
341k 34 546 734

yeah, this, I've never seen anyone use the static method of which you speak, probably for this exact reason... – matt Jan 31 '10 at 11:39

The examples you've seen are flawed, in my view. The standard means of writing a recursive function *doesn't* use or require a static variable in which to store the result; instead, you should use arguments and/or the return value. Use of statics would indeed make them non-rentrant, but they shouldn't be coded that way.

Example using return value (JavaScript):

```
function deepCollectChildText(node) {
    var text, child;

    text = "";
    for (child = node.firstChild; child; child = child.nextSibling) {
        switch (child.nodeType) {
            case 1: // element node, may contain child nodes
                text += deepCollectChildText(child);
                break;
            case 3: // text node
                text += child.value;
                break;
        }
    }
    return text;
}
```

No need for a static. It could have been coded using one, and therefore not been reentrant, but there's no *reason* for it to be.

edited Jan 31 '10 at 11:45

answered Jan 31 '10 at 11:37


T.J. Crowder
328k 45 479 604

Recursive functions are thread safe, to the same extent that normal functions are, in all language architectures I'm aware of that support recursion. Each tread has its own stack, and it is the stack that recursive functions use to store their variables and return addresses.

answered Jan 31 '10 at 11:36

anon

1 Static variables are stored in the data segment, not the stack. – Ignacio Vazquez-Abrams Jan 31 '10 at 11:37

1 What you are saying is that the concept of recursive functions allows for thread-safe code, because each

thread has its own stack. It does not automatically make recursive functions thread-safe however, as you can easily make a recursive function that uses shared state. – Lasse V. Karlsen Jan 31 '10 at 11:38

The OP specifically mentioned ones written using static variables, which makes them non-rentrant. – T.J. Crowder Jan 31 '10 at 11:42

@Ignacio That's why I used the phrase "to the same extent that normal functions are". Also, I have to say too that I don't believe I've ever written a recursive function that uses a static variable, but perhaps that's just me. – anon Jan 31 '10 at 11:46

@Lasse See above – anon Jan 31 '10 at 11:47

IMHO,if static or global variable is used in the function, it might not be thread-safe. Otherwise it is thread-safe.

answered Jan 31 '10 at 11:40

tristan
2,380 2 9 24

Recalling past practices, I've never considered or practised using static variables for recursive functions. Except for constants.

- Re-entrant functions do not modify static variables or shared resources and do not read static non-constants or modifiable shared resources.
- Therefore, Re-entrant => thread-safe, but not vice versa.
- Recursive functions modifying static variables are non-re-entrant.

Therefore, re-entrant recursive functions are thread-safe. While, non-re-entrant recursive functions are not thread-safe, except when access of shared/static resources are effectively constrained by synchronisation boundaries.

Then the following question begs to be answered. If a function modifies a database record, would that make it no longer re-entrant?

I am thinking that as long as the external resource is instantiated per entrantiation, the function is re-entrant. For example, a database record with a unique identity run-key. A new record with a new run-key is generated per entrantiation.

However, is that actually like making a static variable thread-safe? It think it is more like a thread-safe static hash-table that generates a unique key,value pair for each entrantiation and therefore no key,value pairs are shared between entrants.

So, when database records or static resources dispenses unique instances of its resources per entrantiation, a function is effectively re-entrant but I think due to its dependence on thread-safe-guarding of an external shared resource, academics might say it is thread-safe but not re-entrant.

For such an argument, I would beg to differ. For example, for a hypothetical programming language, and its specification does not restrict any of its implementation from using a shared database or a global hash to store variables. So the programmer is unaware of a thread-safe-managed resource being used underneath the implementation of the language. So the programmer goes forth and write a "re-entrant" function, or so he/she thinks. So does that make his/her "re-entrant function" non-re-entrant?

Therefore, my conclusion is, as long as the static/shared resource dispenses a unique instance per entrantiation, a function is re-entrant.

Apologies for coining the terms entrantiation/re-entrantiation, due my lack of knowledge for a better word.

answered Jan 31 '10 at 12:56

Blessed Geek
7,427 5 48 113