

Should a function have only one return statement?

Asked 16 years, 3 months ago Modified 8 years, 11 months ago

Viewed 385k times

779

votes



Locked. This question and its answers are [locked](#) because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

Are there good reasons why it's a better practice to have only one return statement in a function?

Or is it okay to return from a function as soon as it is logically correct to do so, meaning there may be many return statements in the function?

language-agnostic

coding-style

Share


edited Feb 21, 2013 at 2:11

community wiki

10 revs, 7 users 38%

[user50049](#)

25 I don't agree that the question is language agnostic. With some languages, having multiple returns is more natural and convenient than with others. I'd be more likely to complain about early returns in a C function than in a C++ one that uses RAII. – [Adrian McCarthy](#) Jan 15, 2013 at 17:24

3 This is closely related and has excellent answers:
programmers.stackexchange.com/questions/118703/...
– [Tim Schmelter](#) Jul 25, 2014 at 8:03 

language-agnostic? Explain to someone using functional language that he must use one return per function :p
– [Boiethios](#) May 31, 2016 at 13:19

Comments disabled on deleted / locked posts / reviews

50 Answers

Sorted by:

Highest score (default)



1

2

Next

740

votes



I often have several statements at the start of a method to return for "easy" situations. For example, this:

```
public void DoStuff(Foo foo)
{
    if (foo != null)
    {
        ...
    }
}
```

... can be made more readable (IMHO) like this:

```
public void DoStuff(Foo foo)
{
```

```
    if (foo == null) return;


    ...
}
```

So yes, I think it's fine to have multiple "exit points" from a function/method.

Share

edited Mar 11, 2015 at 20:40

community wiki
2 revs, 2 users 90%
Matt Hamilton

-
- 83 Agreed. Although having multiple exit point can get out of hand, I definately think it's better than putting your entire function in an IF block. Use return as often as it makes sense to keep your code readable. – [Joshua Carmody](#) Sep 19, 2008 at 20:20
-
- 172 This is known as a "guard statement" is Fowler's Refactoring. – [Lars Westergren](#) Sep 26, 2008 at 10:22 
-
- 12 When functions are kept relatively short, it is not difficult to follow the structure of a function with a return point near the middle. – [KJAWolf](#) Mar 20, 2009 at 13:48
-
- 21 Huge block of if-else statements, each with a return? That is nothing. Such a thing is usually easily refactored. (atleast the more common single exit variation with a result variable is, so I doubt the multi exit variation would be any more difficult.) If you want a real headache, look at a combination of if-else and while loops (controlled by local booleans), where result variables are set, leading to a final exit point at the end of the method. That is the single exit idea gone mad,

and yes I am talking from actual experience having had to deal with it. – [Marcus Andrén](#) Nov 6, 2009 at 14:51

- 7 'Imagine: you need to do "IncreaseStuffCallCounter" method at the end of 'DoStuff' function. What will you do in this case? :)' -- `DoStuff() { DoStuffInner(); IncreaseStuffCallCounter(); }` – [Jim Balter](#) Jan 31, 2013 at 21:36
-

354 Nobody has mentioned or quoted [Code Complete](#) so I'll do it.
votes



17.1 return

Minimize the number of returns in each routine. It's harder to understand a routine if, reading it at the bottom, you're unaware of the possibility that it returned somewhere above.

Use a *return* when it enhances readability. In certain routines, once you know the answer, you want to return it to the calling routine immediately. If the routine is defined in such a way that it doesn't require any cleanup, not returning immediately means that you have to write more code.

Share

[edited Jun 20, 2020 at 9:12](#)

-
- 64 +1 for the nuance of "minimize" but not prohibit multiple returns. – [Raedwald](#) Mar 9, 2011 at 9:34
-
- 13 "harder to understand" is very subjective, especially when the author does not provide any empirical evidence to support the general claim... having a single variable that must be set in many conditional locations in the code for the final return statement is equally subjected to "unaware of the possibility that the variable was assigned somewhere above int he function! – [Heston T. Holtmann](#) May 5, 2011 at 20:42
-
- 26 Personally, I favour returning early where possible. Why? Well, when you see the return keyword for a given case, you instantly know "I'm finished" -- you don't have to keep reading to figure out what, if anything, occurs afterwards.
– [Mark Simpson](#) Nov 14, 2012 at 15:42
-
- 12 @HestonT.Holtmann: The thing that made *Code Complete* unique among programming books was that the advice *is* backed by empirical evidence. – [Adrian McCarthy](#) May 17, 2013 at 16:08
-
- 9 This should probably the accepted answer, since it mentions that having multiple return points is not always good, yet sometimes necessary. – [Rafid](#) Mar 16, 2014 at 11:07
-

229

votes



I would say it would be incredibly unwise to decide arbitrarily against multiple exit points as I have found the technique to be useful in practice *over and over again*, in fact I have often *refactored existing code* to multiple exit points for clarity. We can compare the two approaches thus:-

```

string fooBar(string s, int? i) {
    string ret = "";
    if(!string.IsNullOrEmpty(s) && i != null) {
        var res = someFunction(s, i);

        bool passed = true;
        foreach(var r in res) {
            if(!r.Passed) {
                passed = false;
                break;
            }
        }

        if(passed) {
            // Rest of code...
        }
    }

    return ret;
}

```

Compare this to the code where multiple exit points *are* permitted:-

```

string fooBar(string s, int? i) {
    var ret = "";
    if(string.IsNullOrEmpty(s) || i == null) return null;

    var res = someFunction(s, i);

    foreach(var r in res) {
        if(!r.Passed) return null;
    }

    // Rest of code...

    return ret;
}

```

I think the latter is considerably clearer. As far as I can tell the criticism of multiple exit points is a rather archaic point of view these days.

Share

edited Jun 6, 2014 at 14:30

community wiki

8 revs, 6 users 93%

[ljs](#)

12 Clarity is in the eye of the beholder - I look at a function and seek a beginning a middle and an end. When the function is small its fine - but when you're trying to work out why something is broken and "Rest of Code" turns out to be non-trivial you can spend ages looking for why ret is – [Murph](#) Sep 11, 2008 at 8:00

7 First of all, this is contrived example. Second, Where does :string ret;" go in the second version? Third, Ret contains no useful information. Fourth, why so much logic in one function/method? Fifth, why not make DidValuesPass(type res) then RestOfCode() separate subfunctions? – [Rick Minerich](#) Mar 20, 2009 at 23:33

25 @Rick 1. Not contrived in my experience, this is actually a pattern I've come across many times, 2. It's assigned in 'rest of code', maybe that's not clear. 3. Um? It's an example? 4. Well I guess it's contrived in this respect, but it's possible to justify this much, 5. could do... – [ljs](#) Mar 21, 2009 at 13:55

5 @Rick the point is that it's often easier to return early than to wrap code up in a huge if statement. It comes up a lot in my experience, even with proper refactoring. – [ljs](#) Mar 21, 2009 at 13:57

- 5 The point of not having multiple return statements is make debugging easier, a far second it is for readability. One breakpoint at the end allows you to see the exit value, sans exceptions. As for readability the 2 shown functions do not behave the same. The default return is an empty string if !r.Passed but the "more readable" one changes this to return null. The author misread that earlier there was a default after only a few lines. Even in trivial examples it is easy to have unclear default returns, something a single return at the end helps enforce. – [Mitch](#) Apr 19, 2013 at 22:16
-

191

votes



I currently am working on a codebase where two of the people working on it blindly subscribe to the "single point of exit" theory and I can tell you that from experience, it's a horrible horrible practice. It makes code extremely difficult to maintain and I'll show you why.

With the "single point of exit" theory, you inevitably wind up with code that looks like this:

```
function()
{
    HRESULT error = S_OK;

    if(SUCCEEDED(Operation1()))
    {
        if(SUCCEEDED(Operation2()))
        {
            if(SUCCEEDED(Operation3()))
            {
                if(SUCCEEDED(Operation4()))
                {
                }
                else
                {
                    error = OPERATION4FAILED;
                }
            }
        }
    }
}
```



```

        }
    }
    else
    {
        error = OPERATION3FAILED;
    }
}
else
{
    error = OPERATION2FAILED;
}
}
else
{
    error = OPERATION1FAILED;
}

return error;
}

```

Not only does this make the code very hard to follow, but now say later on you need to go back and add an operation in between 1 and 2. You have to indent just about the entire freaking function, and good luck making sure all of your if/else conditions and braces are matched up properly.

This method makes code maintenance extremely difficult and error prone.

Share

answered [Aug 31, 2008 at 12:50](#)

community wiki

[17 of 26](#)

-
- 5 @Murph: You have no way of knowing that nothing else occurs after each condition without reading through each one. Normally I would say these kinds of topics are subjective, but this is just clearly wrong. With a return on each error, you are done, you know exactly what happened. – [GEOCHET](#) Sep 15, 2008 at 17:38
-
- 6 @Murph: I've seen this kind of code used, abused and overused. The example is quite simple, as there is no true if/else inside. All that this kind of code needs to break down is one "else" forgotten. AFAIK, this code is in real bad need of exceptions. – [paercebal](#) Sep 17, 2008 at 20:48
-
- 15 You can refactor it into this, keeps its "purity":
`if(!SUCCEEDED(Operation1())) { }else error = OPERATION1FAILED; if(error!=S_OK){
if(SUCCEEDED(Operation2())) { } else error = OPERATION2FAILED; } if(error!=S_OK){
if(SUCCEEDED(Operation3())) { } else error = OPERATION3FAILED; } //etc.` – [Joe Pineda](#) Oct 29, 2008 at 21:27
-
- 6 This code does not have just one exit point: each of those "error =" statements is on an exit path. It's not just about exiting functions, it's about exiting any block or sequence. – [Jay Bazuzi](#) Dec 25, 2008 at 22:29
-
- 6 I disagree that single return "inevitably" results in deep nesting. Your example can be written as a simple, linear function with a single return (without gotos). And if you don't or can't rely exclusively on RAI for resource management, then early returns end up causing leaks or duplicate code. Most importantly, early returns make it impractical to assert post-conditions. – [Adrian McCarthy](#) May 27, 2009 at 16:07
-

72

votes



[Structured programming](#) says you should only ever have one return statement per function. This is to limit the complexity. Many people such as Martin Fowler argue that it is simpler to write functions with multiple return statements. He presents this argument in the classic [refactoring](#) book he wrote. This works well if you follow his other advice and write small functions. I agree with this point of view and only strict structured programming purists adhere to single return statements per function.

Share

edited Sep 5, 2009 at 10:56

community wiki

2 revs, 2 users 89%

cdv


44 Structured programming doesn't say a thing. Some (but not all) people who describe themselves as advocates of structured programming say that. – [wnoise](#) Sep 26, 2008 at 1:44



15 "This works well if you follow his other advice and write small functions." This is the most important point. Small functions rarely need many return points. – user18443 Oct 20, 2008 at 5:44

6 @wnoise +1 for the comment, so true. All "structured programming" says is don't use GOTO. – [oz10](#) Dec 20, 2008 at 0:50

1 @ceretullis: unless it's necessary. Of course it's not essential, but can be useful in C. The linux kernel uses it, and for good reasons. *GOTO considered Harmful* talked about using `GOTO`

to move control flow even when function existed. It never says "never use `GOTO`". – [Esteban Küber](#) Jul 24, 2009 at 17:57

- 1 "are all about ignoring the 'structure' of the code. " -- No, quite the opposite. "it makes sense to say they should be avoided" -- no, it doesn't. – [Jim Balter](#) Jan 5, 2016 at 23:05 

62 votes

 As Kent Beck notes when discussing guard clauses in [Implementation Patterns](#) making a routine have a single entry and exit point ...

"was to prevent the confusion possible when jumping into and out of many locations in the same routine. It made good sense when applied to FORTRAN or assembly language programs written with lots of global data where even understanding which statements were executed was hard work ... with small methods and mostly local data, it is needlessly conservative."

I find a function written with guard clauses much easier to follow than one long nested bunch of `if then else` statements.

Share

answered [Aug 31, 2008 at 9:48](#)

community wiki
[blank](#)

Of course, "one long nested bunch of if-then-else statements" is not the only alternative to guard clauses. – [Adrian McCarthy](#) Mar 15, 2014 at 22:25

@AdrianMcCarthy do you have a better alternative? It would be more useful than sarcasm. – [shinzou](#) Apr 2, 2016 at 18:15

@kuhaku: I'm not sure I'd call that sarcasm. The answer suggests that it's an either/or situation: either guard clauses or long nested bunches of if-then-else. Many (most?) programming languages offer many ways to factor such logic besides guard clauses. – [Adrian McCarthy](#) Apr 4, 2016 at 3:16

61
votes



In a function that has no side-effects, there's no good reason to have more than a single return and you should write them in a functional style. In a method with side-effects, things are more sequential (time-indexed), so you write in an imperative style, using the return statement as a command to stop executing.

In other words, when possible, favor this style

```
return a > 0 ?  
    positively(a):  
    negatively(a);
```

over this

```
if (a > 0)  
    return positively(a);  
else  
    return negatively(a);
```

If you find yourself writing several layers of nested conditions, there's probably a way you can refactor that, using predicate list for example. If you find that your ifs and elses are far apart syntactically, you might want to break that down into smaller functions. A conditional block that spans more than a screenful of text is hard to read.

There's no hard and fast rule that applies to every language. Something like having a single return statement won't make your code good. But good code will tend to allow you to write your functions that way.

Share

answered [Sep 7, 2008 at 18:06](#)

community wiki
[Apocalisp](#)

6 +1 "If you find that your ifs and elses are far apart syntactically, you might want to break that down into smaller functions."
– [Andres Jaan Tack](#) Jun 20, 2010 at 15:19

4 +1, if this is a problem, it usually means you are doing too much in a single function. It really depresses me that this is not the highest voted answer – [Matt Briggs](#) Jul 28, 2011 at 12:17

1 Guard statements don't have any side effects either, but most people would consider them useful. *So there may be reasons to stop execution early even if there are no side effects.* This answer doesn't fully address the issue in my opinion.
– [Maarten Bodewes](#) Jan 2, 2015 at 14:58

@MaartenBodewes-owlstead See ["Strictness and Laziness"](#)
– [Apocalisp](#) Jan 2, 2015 at 16:13

43

votes



I've seen it in coding standards for C++ that were a hang-over from C, as if you don't have RAII or other automatic memory management then you have to clean up for each return, which either means cut-and-paste of the clean-up or a goto (logically the same as 'finally' in managed languages), both of which are considered bad form. If your practices are to use smart pointers and collections in C++ or another automatic memory system, then there isn't a strong reason for it, and it become all about readability, and more of a judgement call.

Share

edited Oct 28, 2010 at 8:36

community wiki

2 revs, 2 users 67%

Pete Kirkham


Well said, though I do believe that it is better to copy the deletes when trying to write highly optimized code (such as software skinning complex 3d meshes!) – [Grant Peters](#) Mar 2, 2009 at 12:56

-
- 1 What causes you to believe that? If you have a compiler with poor optimisation where there is some overhead in dereferencing the `auto_ptr`, you can use plain pointers in parallel. Though it would be odd to be writing 'optimised' code with a non-optimising compiler in the first place. – [Pete Kirkham](#) Mar 2, 2009 at 13:22

This makes for an interesting exception to the rule: If your programming language does not contain something that is

automatically called at the end of the method (such as `try ... finally` in Java) and you need to do resource maintenance you could do with a single return at the end of a method. Before you do this, you should seriously consider refactoring the code to get rid of the situation. – [Maarten Bodewes](#) Dec 29, 2011 at 16:08

@PeteKirkham why is goto for cleanup bad? yes goto can be used badly, but this particular usage is not bad. – [q126y](#) Jun 13, 2016 at 6:20

- 1 @q126y in C++, unlike RAI, it fails when an exception is thrown. In C, it is a perfectly valid practice. See stackoverflow.com/questions/379172/use-goto-or-not – [Pete Kirkham](#) Jun 13, 2016 at 10:56 
-

40
votes



I lean to the idea that return statements in the *middle* of the function are bad. You can use returns to build a few guard clauses at the top of the function, and of course tell the compiler what to return at the end of the function without issue, but returns in the *middle* of the function can be easy to miss and can make the function harder to interpret.

Share

[edited Jul 14, 2010 at 19:15](#)

community wiki
[3 revs, 2 users 60%](#)
[Joel Coehoorn](#)

38
votes

Are there good reasons why it's a better practice to



have only one return statement in a function?



Yes, there are:

- The single exit point gives an excellent place to assert your post-conditions.
- Being able to put a debugger breakpoint on the one return at the end of the function is often useful.
- Fewer returns means less complexity. Linear code is generally simpler to understand.
- If trying to simplify a function to a single return causes complexity, then that's incentive to refactor to smaller, more general, easier-to-understand functions.
- If you're in a language without destructors or if you don't use RAI, then a single return reduces the number of places you have to clean up.
- Some languages require a single exit point (e.g., Pascal and Eiffel).

The question is often posed as a false dichotomy between multiple returns or deeply nested if statements. There's almost always a third solution which is very linear (no deep nesting) with only a single exit point.

Update: Apparently [MISRA guidelines promote single exit](#), too.

To be clear, I'm not saying it's *always* wrong to have multiple returns. But given otherwise equivalent solutions,

there are lots of good reasons to prefer the one with a single return.

Share

edited May 23, 2017 at 12:10

community wiki

6 revs

Adrian McCarthy

-
- 2 another good reason, probably the best these days, to have a single return statement is logging. if you want to add logging to a method, you can place a single log statement that conveys what the method returns. – [inor](#) Feb 13, 2013 at 7:25

How common was the FORTRAN ENTRY statement? See docs.oracle.com/cd/E19957-01/805-4939/6j4m0vn99/index.html. And if you're adventurous, you can log methods with AOP and after-advice – [Eric Jablow](#) Jun 13, 2013 at 0:38

-
- 1 +1 The first 2 points were enough to convince me. That with the second last paragraph. I would disagree with the logging element for the same reason as I discourage deep nested conditionals as they encourages breaking the single responsibility rule which is the main reason polymorphism was introduced into OOP's. – [Francis Rodgers](#) Nov 1, 2014 at 9:55

I'd just like to add that with C# and Code Contracts, the post-condition problem is a nonissue, since you can still use `Contract.Ensures` with multiple return points. – [julealgon](#) Nov 5, 2014 at 13:07

-
- 1 @q126y: If you're using `goto` to get to common cleanup code, then you've probably simplified the function so that there's a single `return` at the end of the clean-up code. So you could say you've solved the problem with `goto`, but I'd

say you solved it by simplifying to a single `return` .

– [Adrian McCarthy](#) Jun 28, 2016 at 22:44

33

votes



Having a single exit point does provide an advantage in debugging, because it allows you to set a single breakpoint at the end of a function to see what value is actually going to be returned.

Share

edited Nov 3, 2014 at 21:58

community wiki

2 revs, 2 users 67%


[Mark](#)

6 BRAVO! You are the **only** person to mention this *objective* reason. It's the reason I prefer single exit points vs. multiple exit points. If my debugger could set a break point on *any* exit point I'd probably prefer multiple exit points. My current opinion is the people who code multiple exit points do it for their own benefit at the expense of those others who have to use a debugger on their code (and yes I'm talking about all you open-source contributors who write code with multiple exit points.)

– [MikeSchinkel](#) Nov 1, 2010 at 10:00


3 YES. I'm adding logging code to a system that's intermittently misbehaving in production (where I can't step through). If the previous coder had used single-exit, it would have been MUCH easier. – [Michael Blackburn](#) Aug 5, 2011 at 17:10

3 True, in debugging it is helpful. But in practice I was able in most cases to set the breakpoint in the calling function, just after the call - effectively to the same result. (And that position



is found on the call stack, of course.) YMMV. – [foo](#) Oct 4, 2011 at 1:49 

This is unless your debugger provides a step-out or step-return function (and each and every debugger does as far as I know), which shows the returned value right *after* is returned.

Changing the value afterwards may be a bit tricky if it is not assigned to a variable though. – [Maarten Bodewes](#) Mar 17, 2014 at 16:13

- 7 I haven't seen a debugger in a long while that would not allow you to set a breakpoint on the "close" of the method (end, right curly brace, whatever your language) and hit that breakpoint regardless of where, or how many, return statements are in the method. Also, even if your function has only one return, that doesn't mean you can't exit the function with an exception (explicit or inherited). So, I think this isn't really a valid point. – [Scott Gartner](#) Nov 21, 2014 at 21:16 
-

19
votes

In general I try to have only a single exit point from a function. There are times, however, that doing so actually ends up creating a more complex function body than is necessary, in which case it's better to have multiple exit points. It really has to be a "judgement call" based on the resulting complexity, but the goal should be as few exit points as possible without sacrificing complexity and understandability.

Share


[edited Jan 8, 2016 at 19:37](#)

community wiki

"In general I try to have only a single exit point from a function"
-- why? "the goal should be as few exit points as possible" --
why? And why did 19 people vote up this non-answer?

– [Jim Balter](#) Jan 5, 2016 at 23:11

@JimBalter Ultimately, it boils down to personal preference.
More exit points *typically* lead to a more complex method
(although not always) and make it more difficult for someone to
understand. – [Scott Dorman](#) Jan 8, 2016 at 19:36

" it boils down to personal preference. " -- In other words, you
can't offer a reason. "More exit points typically lead to a more
complex method (although not always) " -- No, actually, they
don't. Given two functions that are logically equivalent, one with
guard clauses and one with single exit, the latter will have
higher cyclomatic complexity, which numerous studies show
results in code that is more error prone and difficult to
understand. You would benefit from reading the other
responses here. – [Jim Balter](#) Jan 9, 2016 at 0:49 

15
votes

No, because [we don't live in the 1970s any more](#). If your
function is long enough that multiple returns are a problem,
it's too long.



(Quite apart from the fact that any multi-line function in a
language with exceptions will have multiple exit points
anyway.)

Share

edited Jul 14, 2010 at 19:31

14

votes



My preference would be for single exit unless it really complicates things. I have found that in some cases, multiple exist points can mask other more significant design problems:

```
public void DoStuff(Foo foo)
{
    if (foo == null) return;
}
```

On seeing this code, I would immediately ask:

- Is 'foo' ever null?
- If so, how many clients of 'DoStuff' ever call the function with a null 'foo'?

Depending on the answers to these questions it might be that

1. the check is pointless as it never is true (ie. it should be an assertion)
2. the check is very rarely true and so it may be better to change those specific caller functions as they should probably take some other action anyway.

In both of the above cases the code can probably be reworked with an assertion to ensure that 'foo' is never null

and the relevant callers changed.

There are two other reasons (specific I think to C++ code) where multiple exists can actually have a **negative** affect. They are code size, and compiler optimizations.

A non-POD C++ object in scope at the exit of a function will have its destructor called. Where there are several return statements, it may be the case that there are different objects in scope and so the list of destructors to call will be different. The compiler therefore needs to generate code for each return statement:

```
void foo (int i, int j) {  
    A a;  
    if (i > 0) {  
        B b;  
        return ;    // Call dtor for 'b' followed by 'a'  
    }  
    if (i == j) {  
        C c;  
        B b;  
        return ;    // Call dtor for 'b', 'c' and then 'a'  
    }  
    return 'a'      // Call dtor for 'a'  
}
```

If code size is an issue - then this may be something worth avoiding.

The other issue relates to "Named Return Value OptimizatiOn" (aka Copy Elision, ISO C++ '03 12.8/15). C++ allows an implementation to skip calling the copy constructor if it can:

```
A foo () {  
    A a1;  
    // do something  
    return a1;  
}  
  
void bar () {  
    A a2 ( foo() );  
}
```

Just taking the code as is, the object 'a1' is constructed in 'foo' and then its copy construct will be called to construct 'a2'. However, copy elision allows the compiler to construct 'a1' in the same place on the stack as 'a2'. There is therefore no need to "copy" the object when the function returns.

Multiple exit points complicates the work of the compiler in trying to detect this, and at least for a relatively recent version of VC++ the optimization did not take place where the function body had multiple returns. See [Named Return Value Optimization in Visual C++ 2005](#) for more details.

Share

edited Mar 11, 2015 at 20:42

community wiki
2 revs, 2 users 95%
Richard Corden

-
- 1 If you take all but the last dtor out of your C++ example, the code to destroy B and later C and B, will still have to be generated when the scope of the if statement ends, so you

really gain nothing by not have multiple returns. – [Eclipse](#) Oct 30, 2008 at 16:26

- 4 +1 And waaaaay down at the bottom of the list we have **the REAL reason this coding practice exists** - NRVO. However, this is a micro-optimization; and, like all micro-optimization practices, was probably started by some 50 year-old "expert" who is used to programming on a 300 kHz PDP-8, and does not understand the importance of clean and structured code. In general, take Chris S's advice and use multiple return statements whenever it makes sense to.

– [BlueRaja - Danny Pflughoeft](#) Jan 29, 2010 at 21:26 

While I disagree with your preference (in my mind, your Assert suggestion is also a return point, as is a `throw new ArgumentNullException()` in C# in this case), I really liked your other considerations, they are all valid to me, and could be critical in some niche contexts. – [julealgon](#) Nov 5, 2014 at 14:02

This is chock full of strawmen. The question of why `foo` is being tested has nothing to do with the subject, which is whether to do `if (foo == NULL) return; dowork;` or `if (foo != NULL) { dowork; }` – [Jim Balter](#) Jan 9, 2016 at 1:06

11

votes



Having a single exit point reduces [Cyclomatic Complexity](#) and therefore, *in theory*, reduces the probability that you will introduce bugs into your code when you change it. Practice however, tends to suggest that a more pragmatic approach is needed. I therefore tend to aim to have a single exit point, but allow my code to have several if that is more readable.

Share

answered [Sep 2, 2008 at 21:17](#)

Very insightful. Although, I feel that until a programmer knows when to use multiple exit points, they should be constrained to one. – [Rick Minerich](#) Mar 20, 2009 at 23:44

- 5 Not really. The cyclomatic complexity of "if (...) return; ... return;" is the same as "if (...) {...} return;". They both have two paths through them. – [Steve Emmerson](#) Dec 12, 2009 at 20:33
-

11

votes



I force myself to use only one `return` statement, as it will in a sense generate code smell. Let me explain:

```
function isCorrect($param1, $param2, $param3) {  
    $toRet = false;  
    if ($param1 != $param2) {  
        if ($param1 == ($param3 * 2)) {  
            if ($param2 == ($param3 / 3)) {  
                $toRet = true;  
            } else {  
                $error = 'Error 3';  
            }  
        } else {  
            $error = 'Error 2';  
        }  
    } else {  
        $error = 'Error 1';  
    }  
    return $toRet;  
}
```

(The conditions are arbitrary...)

The more conditions, the larger the function gets, the more difficult it is to read. So if you're attuned to the code smell, you'll realise it, and want to refactor the code. Two possible solutions are:

- Multiple returns
- Refactoring into separate functions

Multiple Returns

```
function isCorrect($param1, $param2, $param3) {  
    if ($param1 == $param2) { $error = 'Error 1';  
    if ($param1 != ($param3 * 2)) { $error = 'Error 2';  
    if ($param2 != ($param3 / 3)) { $error = 'Error 3';  
    return true;  
}
```

Separate Functions

```
function isEqual($param1, $param2) {  
    return $param1 == $param2;  
}  
  
function isDouble($param1, $param2) {  
    return $param1 == ($param2 * 2);  
}  
  
function isThird($param1, $param2) {  
    return $param1 == ($param2 / 3);  
}  
  
function isCorrect($param1, $param2, $param3) {  
    return !isEqual($param1, $param2)  
        && isDouble($param1, $param3)  
        && isThird($param2, $param3);  
}
```

Granted, it is longer and a bit messy, but in the process of refactoring the function this way, we've

- created a number of reusable functions,
- made the function more human readable, and
- the focus of the functions is on why the values are correct.

Share

edited Nov 30, 2015 at 9:57

community wiki
3 revs, 3 users 91%
Jrgns

5 -1: Bad example. You have omitted the error message handling. If that would not be needed the `isCorrect` could be expressed just as `return xx && yy && zz;` where `xx`, `yy` and `z` are the `isEqual`, `isDouble` and `isThird` expressions. – [kauppi](#)
Dec 20, 2009 at 7:30

10 I would say you should have as many as required, or any
votes that make the code cleaner (such as [guard clauses](#)).



I have personally never heard/seen any "best practices" say that you should have only one return statement.

For the most part, I tend to exit a function as soon as possible based on a logic path (guard clauses are an excellent example of this).

community wiki

[Rob Cooper](#)10
votes

I believe that multiple returns are usually good (in the code that I write in C#). The single-return style is a holdover from C. But you probably aren't coding in C.

There is no law requiring only one exit point for a method in all programming languages. Some people insist on the superiority of this style, and sometimes they elevate it to a "rule" or "law" but this belief is not backed up by any evidence or research.

More than one return style may be a bad habit in C code, where resources have to be explicitly de-allocated, but languages such as Java, C#, Python or JavaScript that have constructs such as automatic garbage collection and `try..finally` blocks (and `using` blocks in C#), and this argument does not apply - in these languages, it is very uncommon to need centralised manual resource deallocation.

There are cases where a single return is more readable, and cases where it isn't. See if it reduces the number of lines of code, makes the logic clearer or reduces the number of braces and indents or temporary variables.

Therefore, use as many returns as suits your artistic sensibilities, because it is a layout and readability issue, not a technical one.

I have talked about [this at greater length on my blog](#).

Share

edited Dec 4, 2012 at 10:16

community wiki

[4 revs](#)

[Anthony](#)

10

votes



There are good things to say about having a single exit-point, just as there are bad things to say about the inevitable ["arrow"](#) programming that results.

If using multiple exit points during input validation or resource allocation, I try to put all the 'error-exits' very visibly at the top of the function.

Both the [Spartan Programming](#) article of the "SSDSLPeia" and [the single function exit point](#) article of the "Portland Pattern Repository's Wiki" have some insightful arguments around this. Also, of course, there is this post to consider.

If you really want a single exit-point (in any non-exception-enabled language) for example in order to release resources in one single place, I find the careful application of goto to be good; see for example this rather contrived example (compressed to save screen real-estate):

```

int f(int y) {
    int value = -1;
    void *data = NULL;

    if (y < 0)
        goto clean;

    if ((data = malloc(123)) == NULL)
        goto clean;

    /* More code */

    value = 1;
clean:
    free(data);
    return value;
}

```

Personally I, in general, dislike arrow programming more than I dislike multiple exit-points, although both are useful when applied correctly. The best, of course, is to structure your program to require neither. Breaking down your function into multiple chunks usually help :)

Although when doing so, I find I end up with multiple exit points anyway as in this example, where some larger function has been broken down into several smaller functions:

```

int g(int y) {
    value = 0;

    if ((value = g0(y, value)) == -1)
        return -1;

    if ((value = g1(y, value)) == -1)
        return -1;
}

```

```
    return g2(y, value);  
}
```

Depending on the project or coding guidelines, most of the boiler-plate code could be replaced by macros. As a side note, breaking it down this way makes the functions `g0`, `g1`, `g2` very easy to test individually.

Obviously, in an OO and exception-enabled language, I wouldn't use if-statements like that (or at all, if I could get away with it with little enough effort), and the code would be much more plain. And non-arrowy. And most of the non-final returns would probably be exceptions.

In short;

- Few returns are better than many returns
- More than one return is better than huge arrows, and [guard clauses](#) are generally ok.
- Exceptions could/should probably replace most 'guard clauses' when possible.

Share

edited Mar 11, 2015 at 20:44

community wiki
5 revs, 2 users 96%
[Henrik Gustafsson](#)

The example crashes for $y < 0$, because it tries to free the NULL pointer ;-)
– [Erich Kitzmueller](#) Apr 27, 2009 at 6:13

2 opengroup.org/onlinepubs/009695399/functions/free.html "If ptr is a null pointer, no action shall occur." – [Henrik Gustafsson](#) Apr 27, 2009 at 6:41

1 No it will not crash, because passing NULL to free is a defined no-op. It is an annoyingly common misconception that you have to test for NULL first. – [hlovda1](#) Feb 24, 2010 at 20:14

The "arrow" pattern is not an inevitable alternative. This is a false dichotomy. – [Adrian McCarthy](#) Jan 15, 2013 at 18:25

9

votes



You know the adage - **beauty is in the eyes of the beholder**.



Some people swear by [NetBeans](#) and some by [IntelliJ IDEA](#), some by [Python](#) and some by [PHP](#).

In some shops you could lose your job if you insist on doing this:

```
public void hello()  
{  
    if (....)  
    {  
        ....  
    }  
}
```

The question is all about visibility and maintainability.

I am addicted to using boolean algebra to reduce and simplify logic and use of state machines. However, there were past colleagues who believed my employ of "mathematical techniques" in coding is unsuitable, because

it would not be visible and maintainable. And that would be a bad practice. Sorry people, the techniques I employ is very visible and maintainable to me - because when I return to the code six months later, I would understand the code clearly rather seeing a mess of proverbial spaghetti.

Hey buddy (like a former client used to say) do what you want as long as you know how to fix it when I need you to fix it.

I remember 20 years ago, a colleague of mine was fired for employing what today would be called [agile development](#) strategy. He had a meticulous incremental plan. But his manager was yelling at him "You can't incrementally release features to users! You must stick with the [waterfall](#)." His response to the manager was that incremental development would be more precise to customer's needs. He believed in developing for the customers needs, but the manager believed in coding to "customer's requirement".

We are frequently guilty for breaking data normalization, [MVP](#) and [MVC](#) boundaries. We inline instead of constructing a function. We take shortcuts.

Personally, I believe that PHP is bad practice, but what do I know. All the theoretical arguments boils down to trying fulfill one set of rules

quality = precision, maintainability and profitability.


All other rules fade into the background. And of course this rule never fades:

Laziness is the virtue of a good programmer.

Share

edited Mar 11, 2015 at 20:44

community wiki
3 revs, 3 users 80%
Blessed Geek

-
- 1 "Hey buddy (like a former client used to say) do what you want as long as you know how to fix it when I need you to fix it." Problem: it's often not you who 'fixes' it. – [Dan Barron](#) Aug 14, 2013 at 21:54 

+1 on this answer because I agree with where you are going but not necessarily how you get there. I would argue that there are levels of understanding. i.e the understanding that employee A has after 5 years of programming experience and 5 years with a company is very different than that of employee B, a new college graduate just starting with a company. My point would be that if employee A is the only people who can understand the code, it is NOT maintainable and so we should all strive to write code that employee B can understand. This is where the true art is in software. – [Francis Rodgers](#) Nov 1, 2014 at 9:47

-
- 9 votes I lean towards using guard clauses to return early and otherwise exit at the end of a method. The single entry and exit rule has historical significance and was particularly



helpful when dealing with legacy code that ran to 10 A4 pages for a single C++ method with multiple returns (and many defects). More recently, accepted good practice is to keep methods small which makes multiple exits less of an impedance to understanding. In the following Kronoz example copied from above, the question is what occurs in *//Rest of code...?*:

```
void string fooBar(string s, int? i) {  
    if(string.IsNullOrEmpty(s) || i == null) return null;  
  
    var res = someFunction(s, i);  
  
    foreach(var r in res) {  
        if(!r.Passed) return null;  
    }  
  
    // Rest of code...  
  
    return ret;  
}
```

I realise the example is somewhat contrived but I would be tempted to refactor the *foreach* loop into a LINQ statement that could then be considered a guard clause. Again, in a contrived example the intent of the code isn't apparent and *someFunction()* may have some other side effect or the result may be used in the *// Rest of code....*

```
if (string.IsNullOrEmpty(s) || i == null) return null;  
if (someFunction(s, i).Any(r => !r.Passed)) return null;
```

Giving the following refactored function:

```
void string fooBar(string s, int? i) {  
  
    if (string.IsNullOrEmpty(s) || i == null) return null;  
    if (someFunction(s, i).Any(r => !r.Passed)) return null;  
  
    // Rest of code...  
  
    return ret;  
}
```

Share

edited Mar 11, 2015 at 20:46

community wiki
4 revs, 2 users 89%
David Clarke

Doesn't C++ have exceptions? Then why would you return `null` instead of throwing an exception indicating that the argument is not accepted? – [Maarten Bodewes](#) Jan 16, 2013 at 22:24

-
- 1 As I indicated, the example code is copied from a previous answer (stackoverflow.com/a/36729/132599). The original example returned nulls and refactoring to throw argument exceptions wasn't material to the point I was trying to make or to the original question. As a matter of good practice, then yes I would normally (in C#) throw an `ArgumentNullException` in a guard clause rather than return a null value. – [David Clarke](#) Jan 17, 2013 at 0:27
-

7 One good reason I can think of is for code maintenance: you have a single point of exit. If you want to change the

votes



format of the result,..., it's just much simpler to implement.



Also, for debugging, you can just stick a breakpoint there :)

Having said that, I once had to work in a library where the coding standards imposed 'one return statement per function', and I found it pretty tough. I write lots of numerical computations code, and there often are 'special cases', so the code ended up being quite hard to follow...

Share


answered [Aug 31, 2008 at 9:31](#)

community wiki
[OysterD](#)

That does not really make a difference. If you change the type of the local variable that is returned, then you will have to fix all the assignments to that local variable. And it is probably better to define a method with a different signature anyway, as you will have to fix all the method calls as well. – [Maarten Bodewes](#)
Jan 16, 2013 at 22:19

@MaartenBodewes-owlstead - It can make a difference. To name just two examples where you would *not* have to fix all the assignments to the local variable or change the method calls, the function might return a date as a string (the local variable would be an actual date, formatted as a string only at the last moment), or it might return a decimal number and you want to change the number of decimal places. – [nnnnnn](#) Jan 2, 2015 at 14:15

@nnnnnn OK, if you want to do post processing on the output... But I would just generate a new method that does the post processing and leave the old one alone. That's only *slightly* harder to refactor, but you'll have to check if the other calls are

compatible with the new format anyway. But it is a valid reason none-the-less. – [Maarten Bodewes](#) Jan 2, 2015 at 14:20 

7

votes



Multiple exit points are fine for small enough functions -- that is, a function that can be viewed on one screen length on its entirety. If a lengthy function likewise includes multiple exit points, it's a sign that the function can be chopped up further.

That said I avoid multiple-exit functions *unless absolutely necessary*. I have felt pain of bugs that are due to some stray return in some obscure line in more complex functions.

Share

answered [Aug 31, 2008 at 13:38](#)

community wiki

[Jon Limjap](#)

6

votes



I've worked with terrible coding standards that forced a single exit path on you and the result is nearly always unstructured spaghetti if the function is anything but trivial -- you end up with lots of breaks and continues that just get in the way.

Share

answered [Aug 31, 2008 at 9:39](#)

Not to mention having to tell your mind to skip the `if` statement in front of each method call that returns success or not :(– [Maarten Bodewes](#) Jan 16, 2013 at 22:21

6
votes



Single exit point - all other things equal - makes code significantly more readable. But there's a catch: popular construction

```
resulttype res;  
if if if...  
return res;
```

is a fake, "res=" is not much better than "return". It has single return statement, but multiple points where function actually ends.

If you have function with multiple returns (or "res="s), it's often a good idea to break it into several smaller functions with single exit point.

Share

answered [Sep 11, 2008 at 7:39](#)

6

votes



My usual policy is to have only one return statement at the end of a function unless the complexity of the code is greatly reduced by adding more. In fact, I'm rather a fan of Eiffel, which enforces the only one return rule by having no return statement (there's just a auto-created 'result' variable to put your result in).

There certainly are cases where code can be made clearer with multiple returns than the obvious version without them would be. One could argue that more rework is needed if you have a function that is too complex to be understandable without multiple return statements, but sometimes it's good to be pragmatic about such things.

Share

answered [Sep 11, 2008 at 7:48](#)

community wiki

[Matt Sheppard](#)

5

votes



If you end up with more than a few returns there may be something wrong with your code. Otherwise I would agree that sometimes it is nice to be able to return from multiple places in a subroutine, especially when it make the code cleaner.

Perl 6: Bad Example

```

sub Int_to_String( Int i ){
    given( i ){
        when 0 { return "zero" }
        when 1 { return "one" }
        when 2 { return "two" }
        when 3 { return "three" }
        when 4 { return "four" }
        ...
        default { return undef }
    }
}

```

would be better written like this

Perl 6: Good Example

```

@Int_to_String = qw{
    zero
    one
    two
    three
    four
    ...
}
sub Int_to_String( Int i ){
    return undef if i < 0;
    return undef unless i < @Int_to_String.length;
    return @Int_to_String[i]
}

```

Note this is was just a quick example

Share

edited Jun 20, 2020 at 9:12

Ok Why was this voted down? Its not like it isn't an opinion.
– [Brad Gilbert](#) Sep 26, 2008 at 15:29

5

votes



I vote for Single return at the end as a guideline. This helps a **common code clean-up handling** ... For example, take a look at the following code ...

```
void ProcessMyFile (char *szFileName)
{
    FILE *fp = NULL;
    char *pbyBuffer = NULL;

    do {

        fp = fopen (szFileName, "r");

        if (NULL == fp) {

            break;
        }

        pbyBuffer = malloc (__SOME__SIZE__);

        if (NULL == pbyBuffer) {

            break;
        }

        /*** Do some processing with file ***/

    } while (0);

    if (pbyBuffer) {

        free (pbyBuffer);
    }
}
```



```
if (fp) {  
    fclose (fp);  
}  
}
```

Share

edited Mar 11, 2015 at 20:47

community wiki
2 revs, 2 users 97%
Alphaneo

You vote for single return - in C code. But what if you were coding in a language that had Garbage collection and try..finally blocks? – [Anthony](#) Jun 27, 2010 at 17:13

4
votes

 This is probably an unusual perspective, but I think that anyone who believes that multiple return statements are to be favoured has never had to use a debugger on a microprocessor that supports only 4 hardware breakpoints. ;-)



While the issues of "arrow code" are completely correct, one issue that seems to go away when using multiple return statements is in the situation where you are using a debugger. You have no convenient catch-all position to put a breakpoint to guarantee that you're going to see the exit and hence the return condition.

Share

answered Sep 23, 2008 at 23:47

- 5 That's just a different sort of premature optimization. You should never optimize for the special case. If you find yourself debugging a specific section of code a lot, there's more wrong with it than just how many exit points it has. – [Wedge](#) Sep 26, 2008 at 0:44
-

Depends on your debugger too. – [Maarten Bodewes](#) Dec 8, 2012 at 3:23

- 4
votes

 The more return statements you have in a function, the higher complexity in that one method. If you find yourself wondering if you have too many return statements, you might want to ask yourself if you have too many lines of code in that function.

But, not, there is nothing wrong with one/many return statements. In some languages, it is a better practice (C++) than in others (C).

Share

[edited Apr 27, 2009 at 9:56](#)

1

2

Next