Sutter's Mill

Herb Sutter on software development

Feeds: Posts Comments

GotW #2 Solution: Temporary Objects

2013-05-13 by Herb Sutter

Unnecessary and/or temporary objects are frequent culprits that can throw all your hard work—and your program's performance—right out the window. How can you spot them and avoid them?

Problem

JG Question

1. What is a temporary object?

Guru Question

2. You are doing a code review. A programmer has written the following function, which uses unnecessary temporary or extra objects in at least three places. How many can you identify, and how should the programmer fix them?

```
  string find_addr( list<employee> emps, string name ) {
    for( auto i = begin(emps); i != end(emps); i++ ) {
        if( *i == name ) {
            return i->addr;
        }
    }
    return "";
}
```

Do not change the operational semantics of this function, even though they could be improved.

Solution

1. What is a temporary object?

Informally, a temporary object is an unnamed object that you can't take the address of. A temporary is often created as an intermediate value during the evaluation of an expression, such as an object created by returning a value from a function, performing an implicit conversion, or throwing an exception. We usually call a temporary object an "rvalue," so named because it can appear on the "r"ight hand side of an assignment. Here are some simple examples:

```
  widget f();            // f returns a temporary widget object

auto a = 0, b = 1;
auto c = a + b;        // "a+b" creates a temporary int object
```

In contrast, in the same code we have objects like a and c that do each have a name and a memory address. Such an object is usually called an "lvalue," because it can appear on the "l"eft hand side of an assignment.

That's a simplification of the truth, but it's generally all you need to know. More precisely, C++ now has five categories of values, but distinguishing them is primarily useful for writing down the language specification, and you can mostly ignore them and just think about "rvalues" for temporary objects without names and whose addresses can't be taken, and "lvalues" for non-temporary objects that have names and whose addresses can be taken.

2. How many unnecessary temporary objects can you identify, and how should the programmer fix them?

Believe it or not, this short function harbors three obvious cases of unnecessary temporaries or extra copies of objects, two subtler ones, and three red herrings.

The parameters are passed by value.

The most obvious extra copies are buried in the function signature itself:

```
string find_addr( list<employee> emps, string name )
```

The parameters should be passed by const&—that is, const list<employee>& and const string&, respectively—instead of by value. Pass-by-value forces the compiler to make complete copy of both objects, which can be expensive and, here, is completely unnecessary.

> **Guideline:** Prefer passing a read-only parameter by **const&** if you are only going to read from it (not make a copy of it).

Pedantic note: Yes, with pass-by-value, if the caller passed a temporary list or string argument then it could be moved from rather than copied. But I'm deliberately saying "forces the compiler to make a complete copy" here because no caller is realistically going to be passing a temporary list to find_addr, except by mistake.

Non-issue: Initializing with "=".

Next we come to the first red herring, in the for loop's initialization:

```
for( auto i = begin(emps); /*...*/ )
```

You might be tempted to say that this code should prefer to be spelled auto i(begin(emps)) rather than auto i = begin(emps), on the grounds that the = syntax incurs an extra temporary object, even if it might be optimized away. After all, as we saw in GotW #1, usually that extra = means the two-step "convert to a temporary then copy/move" of copy-initialization—but recall that doesn't apply when using auto like this. Why?

Remember that auto always deduces the exact type of the initializer expression, minus top-level const and & which don't matter for conversions, and so… presto! there cannot be any need for a conversion and we directly construct i.

So there is no difference between auto i(begin(emps)) and auto i = begin(emps). Which syntax you choose is up to you, but it depends only on taste, not on temporaries or any other performance or semantic difference.

> **Guideline:** Prefer declaring variables using **auto**. Among other reasons to do so, it naturally guarantees zero extra temporaries due to implicit conversions.

The end of the range is recalculated on each loop iteration.

Another potential avoidable temporary occurs in the for loop's termination condition:

```
for( /*...*/ ; i != end(emps); /*...*/ )
```

For most containers, including list, calling end() returns a temporary object that must be constructed and destroyed, even though the value will not change.

Normally when a value will not change, instead of recomputing it (and reconstructing and redestroying it) on every loop iteration, we would want to compute the value only once, store it in a local object, and reuse it.

> **Guideline:** Prefer precomputing values that won't change, instead of recreating objects unnecessarily.

However, a caution is in order: In practice, for simple inline functions like list<T>::end() in particular used in a loop, compilers routinely notice their values won't change and hoist them out of the loop for you without you having to do it yourself. So I actually don't recommend any change to hoist the end calculation here, because that would make the code slightly more complex and the definition of premature optimization is making the code more complex in the name of efficiency without data that it's actually needed. Clarity comes first:

> *Definition*: **Premature optimization** is when you make code more complex in the name of efficiency without data that it's actually needed.
>
> **Guideline:** Write for clarity and correctness first. Don't optimize prematurely, before you have profiler data proving the optimization is needed, especially in the case of calls to simple inline calls to short functions that compilers normally can handle for you.

The iterator increment uses postincrement.

Next, consider the way we increment i in the for loop:

```
for( /*...*/ ; i++ )
```

This temporary is more subtle, but it's easy to understand once you remember how preincrement and postincrement differ. Postincrement is usually less efficient than preincrement because it has to remember and return its original value.

Postincrement for a class T should normally be implemented using the canonical form as follows:

```
  T T::operator++(int)() {
    auto old = *this; // remember our original value
    ++*this;          // always implement postincr in terms of preincr
    return old;       // return our original value
}
```

Now it's easy to see why postincrement is less efficient than preincrement: Postincrement has to do all the same work as preincrement, but in addition it also has to construct and return another object containing the original value.

> **Guideline:** For consistency, always implement postincrement in terms of preincrement, otherwise your users will get surprising (and often unpleasant) results.

In the problem's code, the original value is never used, and so there's no reason to use postincrement. Preincrement should be used instead. Although the difference is unlikely to matter for a built-in type or a simple iterator type, where the compiler can often optimize away the extra unneeded work for you, it's still a good habit not to ask for more than you need.

> **Guideline:** Prefer preincrement. Only use postincrement if you're going to use the original value.

"But wait, you're being inconsistent!" I can just hear someone saying. "That's premature optimization. You said that compilers can hoist the end() call out of the loop, and it's just as easy for a compiler to optimize away this postincrement temporary."

That's true, but it doesn't imply premature optimization. Preferring ++i does not mean writing more complex code in the name of performance before you can prove it's needed—++i is not more complex than i++, so it's not as if you need performance data to justify using it! Rather, preferring ++i is *avoiding premature pessimization*, which means avoiding writing equivalently complex code that needlessly asks for extra work that it's just going to ignore anyway.

> ***Definition*: Premature pessimization** is when you write code that is slower than it needs to be, usually by asking for unnecessary extra work, when equivalently complex code would be faster and should just naturally flow out of your fingers.

The comparison might use an implicit conversion.

Next, we come to this:

```
    if( *i == name )
```

The employee class isn't shown in the problem, but we can deduce a few things about it. For this code to work, employee likely must either have a conversion to string or a conversion constructor taking a string. Both cases create a temporary object, invoking either operator== for strings or operator== for employees. (Only if there does happen to be an operator== that takes one of each, or employee has a conversion to a reference, that is, string&, is a temporary not needed.)

> **Guideline:** Watch out for hidden temporaries created by implicit conversions. One good way to avoid this is to make constructors and conversion operators **explicit** by default unless implicit conversions are really desirable.

Probably a non-issue: return "".

```
return "";
```

Here we unavoidably create a temporary (unless we change the return type, but we shouldn't; see below), but the question is: Is there a better way?

As written, return ""; calls the string constructor that takes a const char*, and if the string implementation you're using either (a) is smart enough to check for the case where it's being passed an empty string, or (b) uses the small string optimization (SSO) that stores strings up to a certain size directly within the string object instead of on the heap, no heap allocation will happen.

Indeed, every string implementation I checked is smart enough not to perform an allocation here, which is maximally efficient for string, and so in practice there's nothing to optimize. But what alternatives do we have? Let's consider two.

First, you might consider re-spelling this as return ""s; which is new in C++14. That essentially relies on the same implementation smarts to check for empty strings or to use SSO, just in a different function—the literal operator"".

Second, you might consider re-spelling this as return { };. On implementations that are both non-smart *and* non-SSO, this might have a slight advantage over the others because it invokes the default constructor, and so even the most naïve implementation is likely not to do an allocation since clearly no value is needed.

In summary, there's no difference in practice among returning "", ""s, or { }; use whichever you prefer for stylistic reasons. If your string implementation is either smart or uses SSO, which covers all implementations I know of, there's exactly zero allocation difference.

Note: SSO is a wonderful optimization for avoiding allocation overhead and contention, and every modern string ought to use it. If your string implementation doesn't use SSO (as of this writing, I'm looking at you, libstdc++), write to your standard library implementer—it really should.

Non-issue: Multiple returns.

```
    return i->addr;
  return "";
```

This was a second subtle red herring, designed to lure in errant disciples of the "single-entry/single-exit" (SE/SE) persuasion.

I In the past, I've heard some people argue that it's better to declare a local string object to hold the return value and have a single return statement that returns that string, such as writing string ret; … ret = i->addr; break; … return ret;. The idea, they say, is that this will assist the optimizer perform the 'named return value optimization.'

The truth is that whether single-return will improve or degrade performance can depend greatly on your actual code and compiler. In this case, the problem is that creating a single local string object and then assigning it would mean calling string's default constructor *and* then possibly its assignment operator, instead of just a single constructor as in our original code. "But," you ask, "how expensive could a plain old string default constructor be?" Well, here's how the "two-return" version performed on one popular compiler last time I tried it:

- with optimizations disabled: two-return 5% faster than a "return value" string object
- with aggressive optimizations: two-return 40% faster than a "return value" string object

Note what this means: Not only did the single-return version generate slower code on this particular compiler on this particular day, but the slowdown was greater with optimizations turned on. In other words, a single-return version didn't assist optimization, but actively interfered with it by making the code more complex.

In general, note that SE/SE is an obsolete idea and has always been wrong. "Single entry," or the idea that functions should always be entered in one place (at their start) and not with goto jumps from the caller's code directly to random places inside the function body, was and is an immensely valuable advance in computer science. It's what made libraries possible, because it meant you could package up a function and reuse it and the function would always know its starting state, where it begins, regardless of the calling code. "Single exit," on the other hand, got unfairly popular on the basis of optimization ('if there's a single return the compiler can perform return value optimization better'—see counterexample above) and symmetry ('if single entry is good, single exit must be good too') but that is wrong because the reasons don't hold in reverse—allowing a caller to jump in is bad because it's not under the function's control, but allowing the function itself to return early when it knows it's done is perfectly fine and fully under the function's control. To put the final nail in the coffin, note that "single exit" has always been a fiction in any language that has exceptions, because you can get an early exceptional return from any point where you call something that could throw an exception.

Non-issue: Return by value.

Which brings us to the third red herring:

```
string find_addr( /*...*/ )
```

Because C++ naturally enables move semantics for returned values like this string object, there's usually little to be gained by trying to avoid the temporary when you return by value. For example, if the caller writes auto address = find_addr(mylist, "Marvin the Robot");, there will be at most a cheap move (not a deep copy) of the returned temporary into address, and compilers are allowed to optimize away even that cheap move and construct the result into address directly.

But what if you did feel tempted to try to avoid a temporary in all return cases by returning a string& instead of string? Here's one way you might try doing it that avoids the pitfall of returning a dangling reference to a local or temporary object:

```
  const string& find_addr( /* ... */ ) {
    for( /* ... */ ) {
        if( /* found */ ) {
            return i->addr;
        }
    }
    static const string empty;
    return empty;
}
```

To demonstrate why this is brittle, here's an extra question:

For the above function, write the documentation for how long the returned reference is valid.

Go ahead, we'll wait.

Done? Okay, let's consider: If the object is found, we are returning a reference to a string inside an employee object inside the list, and so the reference itself is only valid for the lifetime of said employee object inside the list. So we might try something like this (assuming an empty address is not valid for any employee):

> "If the returned string is nonempty, then the reference is valid until the next time you modify the employee object for which this is the address, including if you remove that employee from the list."

Those are very brittle semantics, not least because the first (but far from only) problem that immediately arises is that the caller has no idea which employee that is—not only doesn't he have a pointer or reference to the right employee object, but he may not even be able to easily figure out which one it is if two employees could have the same address. Second, calling code can be notoriously forgetful and careless about the lifetimes of the returned reference, as in the following code which compiles just fine:

```
  auto& a = find_addr( emps, "John Doe" );  // yay, avoided temporary!
emps.clear();
cout << a;                                  // oops
```

When the calling code does something like this and uses a reference beyond its lifetime, the bug will typically be intermittent and very difficult to diagnose. Indeed, one of the most common mistakes programmers make with the standard library is to use iterators after they are no longer valid, which is pretty much the same thing as using a reference beyond its lifetime; see GotW #18 for details about the accidental use of invalid iterators.

Summary

There are some other optimization opportunities. Ignoring these for now, here is one possible corrected version of find_addr which fixes the unnecessary temporaries. To avoid a possible conversion in the employee/string comparison, we'll assume there's something like an employee::name() function and that .name() == name has equivalent semantics.

Note another reason to prefer declaring local variables with auto: Because the list<employee> parameter is now const, calling begin and end return a different type—not iterators but const_iterators—but auto naturally deduces the right thing so you don't have to remember to make that change in your code.

```
  string find_addr( const list<employee>& emps, const string& name ) {
    for( auto i = begin(emps);  i != end(emps); ++i ) {
        if( i->name() == name ) {
            return i->addr;
        }
    }
    return "";
}
```

Acknowledgments

Thanks in particular to the following for their feedback to improve this article: "litb1," Daan Nusman, "Adrian," Michael Marcin, Ville Voutilainen, Rick Yorgason, "kkoehne," and Olaf van der Spek.

Posted in GotW | 59 Comments

59 Responses

Kenneth Ho *on 2013-11-03 at 9:48 pm*

0

0

i
Rate This

Hi Sutter,

I failed to find a more appropriate article to post this question, so here it goes.

Why doesn't std::min/max take URefs? I imagine something along the lines of the following would be reasonable:

```
template <typename T>
T min(T&& v1, T&& v2)
{
  return v1 < v2 ? std::forward<T>(v1) : std::forward<T>(v2);
}
```

The Bogus SE/SE Rule | Bulldozer00's Blog *on 2013-07-25 at 1:03 am*

0

0

i
Rate This

[…] Single Entry / Single Exit (SE/SE) "rule" of programming. I wish I had this eloquent Herb Sutter treatise on hand when it […]

GeoffW *on 2013-05-23 at 8:56 pm*

0

0

i
Rate This

@Herb Sutter: Thanks for your answer. I had not considered locality. Saying "a wash" makes It is easier to understand why libstdc++ may not have rushed into using it – of course the status with C++11 changes that. I note also that consistent use of iterators over operator[] (one of the places where the extra test is potentially significant) would help even if the optimiser should miss it. I've learned to be careful about assuming what optimisers may do. I'm looking forward to a review of GotW#33 and discussing inline which has given me some grief lately (it's often trying to do double duty, normal inline and macro/code replacement, the latter of which can be problematic in the face of optimiser choices).

Herb Sutter *on 2013-05-23 at 3:41 pm*

0

0

i

Rate This

@GeoffW: I don't know of SSO specific studies offhand, but it's well known from studies that small strings dominate most applications, and that SSO optimizes two specific things that are well-known to be important: locality, and allocation (for allocation it's both straight cost plus often poor scaling, though scaleable allocators are getting more common).

When Dinkumware switched from COW (which still had bugs) to SSO back around 2000, P.J. Plauger told me about performance measurements he had made that justified SSO, and also that when he made the switch he got no performance feedback from customers. You know that if there were any significant performance regressions in some code patterns then customers would report it. So this means switching from COW to SSO was "a wash" (his words) or better across a very broad set of customer code bases. (And of course switching away from COW put him in a good place because COW was made non-conforming in C++11.)

Finally, if you want a little more, remember that: (a) basic_string is a template which means all of its functions are inline; and (b) you usually perform more than just one operation on a string in a given function. So most of the time I would assume that optimizers routinely hoist/eliminate repeated code like isInternal ? useInternal : useExternal checks without even trying very hard. That's low-hanging fruit for a modern optimizer.

GeoffW *on 2013-05-23 at 2:04 pm*

0

0

i
Rate This

The more I look at SSO the more I wonder just how effective it is – but I'm having trouble finding any discussion of actual testing. It sounds great – no allocation overheads for some percentage of strings – but the cost is that almost every interaction with the string object comes prefixed with (isInternal ? useInternal : useExternal). Yes this is very minor when taken alone, but when added to the extra cost of move, the extra space consumed by even empty strings (think vectors and lists of objects containing strings), it must add up over the life of the object and application. Testing the difference between COW and non-COW was relatively simple, but testing the advantages of SSO is much less so (and may vary widely between applications).

GeoffW *on 2013-05-23 at 1:49 pm*

0

0

i
Rate This

@Thomas Gahr: The single pointer implementation points to a structure that embeds the counter, size etc. in the allocated buffer – essentially just your typical pimpl pattern. There are many examples of this – for an easy to read version see the test harness that went with the old GotW#45 (COW_AtomicInt2) – more realistic implementations optimise the empty string case.

Thomas Gahr *on 2013-05-23 at 7:04 am*

0

0

i
Rate This

@GeoffW: FWIW: as far as I can see counter-COW argument boils down to exception safety since now any non-const access to the string (like operator[]) can throw std::bad_alloc as it mandates a deep copy of the data. I know your question was mainly about SSO, I just wanted to complete my last comment.

Thomas Gahr *on 2013-05-23 at 6:48 am*

0

0

i
Rate This

@GeoffW: I don't think sizeof(void*) std::string is doable no matter if you use SSO or COW – if we take COW, you'll need one pointer for the data and one pointer for the reference count (or linked list, whichever you prefer) at least. You'll also want to store the size of the string or you're back to ::strlen for implementing .size(). If you take the "dumb" implementation of a string without any optimization – always allocate and memcpy data – you'll need the data-pointer and the size member so you're at 2*sizeof(void*) at least. Or am I wrong?
Either way, I'd be interested in links to discussions about advantages/disadvantages of SSO and/or COW, too.

GeoffW *on 2013-05-21 at 8:56 pm*

1

0

i
Rate This

A question about the short string optimisation comments made in this article. You say that "SSO is a wonderful optimization", but offer no references in support of this. Are there articles around that demonstrate the advantages of SSO in a realistic manner? I mean SSO looks good in theory, but so does copy-on-write. Is it proven to be worth blowing out the size of a string object from a single pointer to something that is some multiple of that just to implement SSO? In particular I am wondering if the perceived advantages of SSO may be less as more code takes advantage of C++11 move semantics in their interfaces.

GeoffW *on 2013-05-20 at 7:38 pm*

0

1

i
Rate This

I think @Adrian makes some good points about single-exit. It was never about optimisation, or structured programming rules (or not for me), it was about code clarity. In short, simple functions I am happy to have multiple exits, it's all there in front of you and your are unlikely to miss them when you come back to maintain the function. In longer or more complex functions multiple-exit points can become a problem for maintenance – and any performance advantages in those situations are dubious and would have to be demonstrated. (Of course we all avoid longer and more complex functions when we can, but they do still happen.)

Herb Sutter *on 2013-05-19 at 8:14 am*

0

0

i
Rate This

@Ahmed: I used to agree with you — the original GotW and Exceptional C++ said "hoist end()". I switched to the current recommendation at Bjarne's urging because compilers have for a long time routinely inlined the container.end() case in particular. Re (1), no you don't need to inline both find_addr and end, just end — find_addr is its own function. Re (2), I agree one should write the clearest code first, so the extra variable for end is arguably less clear. But yes, range-for and std::find are the preferred answer anyway, see GotW #3.

Ahmed Charles *on 2013-05-19 at 3:05 am*

1

0

i
Rate This

I disagree with both points given which suggest that the end iterator being recomputed each loop should be preferred.

1) After seeing Chandler's talk on Optimizations at C++Now, I'm significantly less likely to think that compilers will optimize this sort of code into 'the right thing'. It would require inlining find_addr and end, while hoping that emps is a stack local in the caller.

2) Ignoring optimizations and compiler quality: For the same reasons that using algorithms is better than a normal for loop, because they are constrained and therefore provide more information to the reader, a for loop with the end calculated once provides more information to the reader, in that it guarantees that end isn't changing (or if it is, that it's likely to be a bug). I also don't view this as a premature optimization, because it should simply be idiomatic to write code with the begin and end iterators assigned to variables in the for loop initializer, much like writing ++i is idiomatic. Granted, it should be idiomatic to write this as a range based for loop or std::foreach with a lambda, rather than a normal for loop. And with C++14, both of these are probably equivalent or less typing and they both cause end to be called only once.

Herb Sutter *on 2013-05-17 at 11:51 am*

2

0

i
Rate This

@Robert: That's not the same example as in the article. :) I agree your two lines are equivalent, and in both cases the calling code can easily see the bug locally in code review just by knowing the return type of the function (by value). What's different to me in find_addr is that we'd be returning a reference that refers, not "directly within this here object I'm returning," but "deep within some data structure plus I'm not going to tell you exactly which node it is either." The article was doing the latter, and there's no way to check the calling code's correctness during code review without carefully reading the English documentation of find_addr to figure out if the reference was still valid. Doing the former, returning a reference directly into this visible object the caller knows about, is much less problematic IMO and does not require English documentation analysis to spot the bug.

@tivrfoa: I just compared using "no optimizations" vs. "max optimizations" compiler flags.

tivrfoa *on 2013-05-17 at 7:21 am*

0

0

i

Rate This

Hello. "with aggressive optimizations". How are these "aggressive optimizations"? Could someone show an example? Thank you.

Róbert Dávid *on 2013-05-17 at 1:14 am*

0

1

i
Rate This

@GregM:

```
1 | std::get<0>()
```

(blog ate my pointy braces) isn't a member function of std::tuple, still isn't returning by value..
@Herb: Wait a minute. Are you saying that

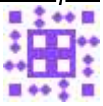
```
1 | auto& x = function_that_creates_widget_and_returns_by_value().get_name();
```

is 'good', but

```
1 | auto& x = get_widget_name(function_that_creates_widget_and_returns_by_value(
```

isn't? I so far failed to create any example where a get(xx) construct can have a dangling reference what is fixed just by replacing that call to a xx.get() construct, that's why I think the two are equivalent, thus should use the same guidelines for return type.

Rick Yorgason *on 2013-05-16 at 6:32 pm*



1

1

i
Rate This

@Herb But in practice, I've never seen it cause much of a problem. Any time a function is returning a &, whether it's a getter or a function like yours, the caller always knows one thing: the return value is valid if used immediately.

So if you're returning a const&, I know that both of these are safe:
A) string s = find_addr(emps, "Rick");
B) if(find_addr(emps, "Rick").find("Toronto") != string::npos) {}

But if you're returning a value, one of those is less efficient.

I agree that hanging references/iterators is a very common problem, but every time I've encountered it, the caller made a conscious decision to store a reference rather than a value, including in the example you provided.

Herb Sutter *on 2013-05-16 at 6:15 am*

0

0

i
Rate This

@kkoehne: Which ironically was the one Ville quoted. :) Thanks.

kkoehne *on 2013-05-15 at 10:27 pm*

0

0

i
Rate This

@Herb: You only fixed it in one place. Two paragraphs below there still is "So there is no difference between auto i{begin(emps)}"

Herb Sutter *on 2013-05-15 at 1:04 pm*

0

0

i
Rate This

@Ville: Good catch, I meant (). Fixed, thanks.

@Rick: & parameters are much safer because they do fall into the structured lifetime bucket — as long as the caller holds a valid reference at the time he calls the function (and modulo aliasing) the reference is guaranteed to stay valid for the lifetime of the function. The same is not true when you return a pointer or reference, or store a pointer or reference on the heap or anywhere else non-local, because by definition that's unstructured.

Herb Sutter *on 2013-05-15 at 1:00 pm*

0

0

i
Rate This

@Adrian, @Michael: OK, I've added that "helping optimization" was another motivation for single exit and added a bit more discussion about RVO and how the measurement I presented demonstrated that single-return actually harmed optimization.

Herb Sutter *on 2013-05-15 at 12:50 pm*

1

0

i
Rate This

@Róbert Dávid: No, returning a & can make sense. In the cases you mention, the lifetime issue is considerably simpler because you're returning a reference guaranteed to be valid for the lifetime of the same object that is producing it, which is very different and much easier to reason about. In the case in the article, it's a function returning a reference into an unknown node inside the list data structure; in the case of a getter like auto& s = mywidget.get_name(), it's a function returning a reference that is good for as long as the known named mywidget object is alive.

GregM *on 2013-05-15 at 9:06 am*

1

1

i
Rate This

Robert,

"By that logic, no getter function should return a const reference, including vector::operator[], map::at(), std::get(), etc.! The fact that find_addr is free-standing doesn't make a difference compared to the usual member getters"

I think it is much more obvious that [], at(), and get() member functions return data whose lifetime is defined by the container than it is that a standalone function returns data whose lifetime is defined by one of its parameters.

GregM *on 2013-05-15 at 9:03 am*

0

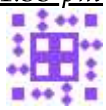
0

i
Rate This

"Yes, checking !*p is the "check for empty string" I was talking about."

Thanks. I wasn't sure whether it was "checking for !*p" or "checking for 0 == strlen(p)".

Rick Yorgason *on 2013-05-14 at 4:55 pm*



0

0

i
Rate This

I'm not sure it's fair to demonize return by const& while preferring const& parameters. Both of these can lead to brittle code: in the former case, it's because of hanging references, and in the latter case it's because of aliasing. But often the brittleness is worthwhile, especially if you have large objects or if it's likely that the return value will be chained or passed into another function.

However, it's very rare that I use that static string technique. Usually if I'm returning by reference, it's a caller error to ask for a reference that doesn't exist, and I'll throw an exception.

Herb Sutter *on 2013-05-14 at 2:47 pm*

0

0

i
Rate This

@Daan: Yes, but part of the question was not to mess with the basic shape of the function. :) We're focusing on temporary/extra objects in this one.

Gonzalo BG *on 2013-05-14 at 1:07 pm*

0

0

i
Rate This

"So I actually don't recommend any change to hoist the end calculation here, because that would make the code slightly more complex […]"

Does

```
1  for(auto i = begin(emp), e = end(emp); i != e; ++i) {
2    //...
3  }
```

make the code more complex? The list::end() iterator is never invalidated. I think that what you propose, i.e. calling the function end() at every loop iteration (and hoping for the compiler to optimize it), fits into your description of premature pessimization.

Ville Voutilainen *on 2013-05-14 at 12:47 pm*

1

1

i
Rate This

The blog ate the code. It will give you

```
1  initializer_list<list<employee>::iterator>
```

Ville Voutilainen *on 2013-05-14 at 12:46 pm*

0

0

i
Rate This

Oops, meant to say the former will give you an initializer_list. That it will do. :)

Ville Voutilainen *on 2013-05-14 at 12:45 pm*

0

0

i
Rate This

Uh oh. "So there is no difference between auto i{begin(emps)} and auto i = begin(emps). "
Sure there is – the latter will give you an initializer_list<list::iterator>>. Which is probably not what you want.

Olaf van der Spek *on 2013-05-14 at 12:51 am*

0

3

i
Rate This

@Michael A safety belt does not affect your car's performance, returning by value does.

Martin *on 2013-05-14 at 12:15 am*

1

0

i
Rate This

I still prefer:

```
1 for(auto i=begin(x), e=end(x); i!=e; ++i)
```

It's not so much about the optimization, it's that the semantics now exactly match range based for.

Rvalue and Lvalue | oraclechang *on 2013-05-13 at 6:21 pm*

0

0

i
Rate This

[…] GotW #2 Solution: Temporary Objects (herbsutter.com) […]

Michael Marcin *on 2013-05-13 at 4:27 pm*

0

0

i
Rate This

I've heard it's harder/impossible for the compiler to do RVO with multiple exit functions, is this true?

Róbert Dávid *on 2013-05-13 at 4:14 pm*

0

0

i
Rate This

I disagree with the return type argument. By that logic, no getter function should return a const reference, including vector::operator[], map::at(), std::get(), etc.! The fact that find_addr is free-standing doesn't make a difference compared to the usual member getters – it's the same deal as with std::get().

```
1  auto& a = find_addr( emps, "John Doe" );  // yay, avoided temporary!
2  emps.clear();
3  cout << a;                                // oops
```

is almost the same as

```
1  auto& a = emps.front().addr;
2  emps.clear();
3  cout << a;                                // oops
```

and even,

```
1  vector<int> emps = { 0, 1, 2, 3 };
2  int& a = emps[1];
3  emps.clear();
4  cout << a;                                // oops
```

Herb Sutter *on 2013-05-13 at 1:51 pm*

0

0

i
Rate This

@litb1: Good point, clarified. I was sure that somewhere over the years I already fixed that to say "temporary or extra objects" but perhaps it was in a different GotW. Thanks.

Herb Sutter *on 2013-05-13 at 1:48 pm*

0

0

i
Rate This

@GregM: Yes, checking !*p is the "check for empty string" I was talking about.

Herb Sutter *on 2013-05-13 at 1:46 pm*

2

0

i
Rate This

@Thomas: Right, either check-for-empty or SSO is sufficient by itself, and all implementations I know of do at least one of those. It's only if you don't check for empty and you don't do SSO that you can get an allocation with return "";.

Having said that, since you're looking into libstdc++'s basic_string, there's actually something even worse in there — instead of SSO, when I tried it on the weekend it looked like gcc 4.8 libstdc++ was still doing COW (copy on write). COW has long been a bad idea and is now actively nonconforming to C++11, which deliberately made COW basic_string implementations nonconforming. So the fix you want should probably be phrased something like "please rip out brittle COW to conform to C++11 and replace it with modern SSO like everyone else did years ago." :) I've been bellyaching about this for years, see 1999's "Optimizations That Aren't (in a Multithreaded World)" http://www.gotw.ca/publications/optimizations.htm .

(Obligatory disclaimer: Yes, VC++ has way more conformance issues than GCC right now, and I'm pointing those out too. Just sayin' that this is also a decade-long overdue fix to libstdc++.)

Adrian *on 2013-05-13 at 12:52 pm*

3

10

i
Rate This

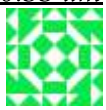
@Herb: I shouldn't have mentioned "structured programming" as that brings other debates into it.

My point was to contest your claim that single exit style exists only because of symmetry arguments with single entry. This is simply not true, and, in fact, I've never heard the symmetry argument made before.

I argue that single exit remains a valuable, useful practice that is neither "errant" nor "obsolete." Single exit code is almost always easier to understand and reason about, and it provides a chokepoint to assert invariants and post conditions. This has nothing to do with resource management or variable lifetimes–we all agree that RAII is the right thing to do. Nor is the argument for single exit specific to languages like C. The cases where early returns help, even in C++, are minimal, and in many, many cases, they introduce complexity and reduce robustness.

@Michael Gmelin: Single-exit does not inherently lead to deep nesting, and multiple return code is generally more fragile when code must be modified to meet new requirements.

Michael Gmelin *on 2013-05-13 at 10:33 am*



3

0

i
Rate This

@Herb: Couldn't agree more. Single-exit tends to lead to deep nesting, hard to read and easy to break code. In C it's common to use goto for proper clean up and reduce nesting (therefore doing what C++ does automatically explicitly and as a result of it have single exit). Been there, it doesn't make sense in C++ at all – especially new/delete free C++14 code.

Maybe calling it "early return" isn't helpful though, since that implies it's something extraordinary. To me, returning as soon as a final result is available is quite common these days.

Herb Sutter *on 2013-05-13 at 10:17 am*

7

0

i
Rate This

@Adrian: Early return is not a violation of structured programming. This is exactly at the heart of the misconception that single entry (which is essential to structured programming) has anything at all to do with single exit (which is not). Structured means simply that lifetimes (such as of function calls and local variables) nest properly, like Russian dolls — which jumping into a function would violate, but returning early or throwing does not.

Having said that, "single exit" matters, but for different reasons(!), in languages like C where you don't get stack lifetimes and automatic cleanup, and you typically have a cascade of goto labels and a single return at the end to make sure you got it right. For example, see http://www.c2.com/cgi/wiki?SingleFunctionExitPoint . That's a limitation of C specifically, not an inherent benefit of single exit or a generally symmetry with single entry which is important in all languages for entirely different reasons.

Ambarish Sridharanarayanan *on 2013-05-13 at 8:34 am*

0

0

i
Rate This

I'd always thought of something like SSO as a no-brainer, but Andrew Koenig's tale has made me stop and think:

http://www.drdobbs.com/cpp/is-optimization-immoral/240151916

Dave Harris *on 2013-05-13 at 8:06 am*

2

0

i
Rate This

It seems to me that changing the argument from by-value to by-reference changes the operational semantics of this function. In the original, the function gets a local copy of the list, which cannot be affected by anything outside of the function. The by-reference list can be affected by other code. That is mostly likely if the code is multi-threaded, but it can also happen if the list is accessible from a static variable and some of the functions called modify it.

(It also makes it harder for the compiler to optimise things like end(). For example, if the (*i == name) line invokes code that the compiler can't see, then it's harder for the compiler to be sure that it does not modify the list through some alias.)

(I'd also hesitate to say emps is a temporary variable, given that it has a name and its address can be taken. Yes, it is an unnecessary copy that you would want to avoid, but it surely isn't a temporary variable.)

The (*i == name) does not need a temporary if employee has an operator==() that accepts a string. I would have thought that were more common than being able to convert an employee to a string or vice versa.

Adrian *on 2013-05-13 at 7:53 am*

3

11

i
Rate This

""Single exit," on the other hand, got unfairly popular on the basis of symmetry" [citation needed]

Single exit popular because it makes code easier to reason about. It provides more places to assert your invariants, which helps in ensuring your code is correct. Early returns (and other violations of structured programming) make invariant checking impossible.

Exceptions are exceptional–that is, they usually mean your invariant is violated anyway, so it makes sense to have those skip your invariant checking. But early returns in non-exceptional cases create complexity by creating more paths that have to be analyzed and understood.

In the benchmark I just threw together, early returns provided only a 10% increase in speed with aggressive compiler optimizations, so I'd accept an early return as an optimization in performance-critical code. I'd also except it in the rare case that it does clarify rather than confuse. But in general, early returns are a premature optimization: more complex code without data supporting a performance improvement.

litb1 *on 2013-05-13 at 7:41 am*

8

1

i
Rate This

I recommend to change

"The most obvious temporaries are buried in the function signature itself:"

To avoid saying "temporaries", or to clarify your definition of "temporary". Both of these by-value parameters have a name, their lifetime last the entire functions activation, and their address can be taken. Or did I misunderstand your descriptions?

Daan Nusman *on 2013-05-13 at 7:20 am*

6

0

i
Rate This

std::optional as return type might be useful here (this is a C++14 GotW after all).

Chris *on 2013-05-13 at 6:53 am*

0

2

i
Rate This

@zahirtezcan, That's exactly what I was thinking. I'd probably change the ending to use the ternary conditional operator, though.

Albert *on 2013-05-13 at 6:52 am*

2

0

i
Rate This

By the way, concerning the other advice "Guideline: Prefer preincrement. Only use postincrement if you're going to use the original value. ", I always use preincrement. If I need post increment, I explicity do the copy and then call the preincrement :

```
1  ++v;
2  //or if post increment needed :
3  v1 = v;
4  ++v;
```

I think the intention is clear doing so and avoids a reviewer to wonder if a post increment has been wanted by the coder or is an inattention.

Albert *on 2013-05-13 at 6:48 am*

5

0

i
Rate This

Hello,
Thank you for this new *modern* gotw source of thought and learning as usual.

It happend that this morning I also read the article from Andrew Koenig about *subtleties of Aliasing* (http://www.drdobbs.com/cpp/some-subtleties-of-aliasing/240154618) which has a slightly different conclusion about the rule : "Guideline: Prefer passing read-only parameters by const& instead of by value. "

For a long, I had the same conclusion as Mr Sutter about prefering const reference than value. But, since a time I changed my mind.
I observed that my interfaces have more and more 'light' objects as parameters (*). For instance, a function like find_addr would probably take an iterator pair (possibly wrapped in a range) rather than an explicit container.
Even for the string parameter, I am higly tempted to pass a value. To my mind, it makes the function more loosely of what happen to the string outside the function which, I think, enforces the reliability of the find_addr function. const reference does only write in the contract that the supplyer does not change the value of the string but tells nothing about the customer. Yesterday, all multithread seemed so far away, now it looks as though they're here to stay ;)
(*) well, my remarks, of course, concern arguments that have a 'value' semantic that it to say those that are mainly substituable by any instance with the same value. Classes providing a 'reference' semantic (that is distinct only by their adress not by their internal state) are of course systematically passed by reference (and even more so that the semantics of reference is generally incompatible with the copy/move ones). In fact, potentially big (i.e. 'costly') value-typed object I manipulate tend to have helpers (like iterator) that are more light and are often better candidates for function since they isolate a facet of the object.

Regards,

Alb

Thomas Gahr *on 2013-05-13 at 6:34 am*

0

0

i
Rate This

Huh. You just sent me off reading through lilbstdc++'s string sources and indeed they don't use sso (or I just didn't understand the source code correctly), though some posts I found on the interwebs suggest they at least did so a few years ago [1].
If anyone could point me to some post with a discussion about when/why they changed, I'd be very interested. I found [2] (pre-sso) where it is raised that a decent allocator can bring enough advantage to not require sso – though they're using the default allocator in std::string – so no speedy small strings for g++ folks :(

However – and this is relevant for the problem posed here: they're using the null object pattern for the empty string so return std::string() is not expensive.

[1] https://bugzilla.redhat.com/show_bug.cgi?id=197718#c30
[2] http://gcc.gnu.org/ml/libstdc++/2005-11/msg00013.html

GregM *on 2013-05-13 at 6:17 am*

2

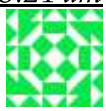
0

i
Rate This

In summary, there's no difference in practice among returning "", ""s, or { }; use whichever you prefer for stylistic reasons. If your string implementation is either smart or uses SSO, which covers all implementations I know of, there's exactly zero allocation difference.

What about performance difference? Don't the first two forms require a strlen call, or are there string implementations that check the first character for NUL and thus skip the strlen?

Michael Gmelin *on 2013-05-13 at 5:24 am*



1

1

i
Rate This

@minhingent:
http://martinfowler.com/refactoring/catalog/replaceNestedConditionalWithGuardClauses.html

zahirtezcan (@zahirtezcan) *on 2013-05-13 at 5:22 am*

2

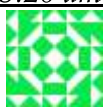
3

i
Rate This

what about using polymorphic lambdas for find_if algorithm?

```
auto endEmp = end(emps);
auto found = std::find_if(begin(emps), endEmp, [&](auto& emp) { return emp.n
if(found != endEmp)
    return found->addr;
else
{
    /*return empty*/
}
```

Michael Gmelin *on 2013-05-13 at 5:20 am*



3

0

i
Rate This

I would probably write

```
for( const auto& e : emps  )
```

too, since this is the "safe default" way of writing range based for loops.

But like you said, in such a short function I wouldn't bring it up either (the signature and the code are right next to each other). In a longer function I would probably ask to change it to const to reduce the probably of maintenance disaster though.

minhingent *on 2013-05-13 at 5:19 am*

0

6

i
Rate This

Regarding single-entry/single-exit, what about using a break statement when the employee is found?

kjellkod *on 2013-05-13 at 5:14 am*

6

0

i
Rate This

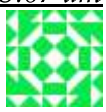
@Michael. Good comment. It makes the for-loop much easier to read.

nitpick warning

```
1 for( const auto& e : emps  )
```

Using explicit "const" in the loop is as you pointed is out not necessary but highlights even more that it is const without really affecting the readability. I would prefer that personally but would not bring it up as an issue at a code review.

Michael Gmelin *on 2013-05-13 at 5:07 am*



6

1

i
Rate This

That's like saying, that wearing a safety belt is premature pessimization. ;)

Olaf van der Spek *on 2013-05-13 at 4:57 am*

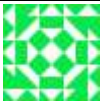
7

14

i
Rate This

Returning by value is premature pessimization. It also disallows a mutable reference or pointer to be returned. Ideally this function would be named find_employee and return a const employee*. That also avoids the problem of differentiating between employee not found and employee without address.

Michael Gmelin *on 2013-05-13 at 4:41 am*



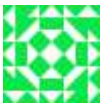
1

2

i
Rate This

-loop

Michael Gmelin *on 2013-05-13 at 4:40 am*



41

3

i
Rate This

Using C++11/14 I recommend my team to use a range based for loop whenever possible.

```
string find_addr( const list<employee>& emps, const string& name ) {
    for( auto& e : emps  ) {
        if( e.name() == name ) {
            return e.addr;
        }
    }
    return "";
}
```

That way the implementer doesn't have to think hard about optimizing away the end, pre- or post-increment etc. It should be safe to assume that the compiler knows how to optimize a range based for loop (note that const is not necessary, since the input is const already).

Comments are closed.