GotW #91 Solution: Smart Pointer Parameters

Herb Sutter 2013-06-052013-12-03 9 Minutes

NOTE: *Last year, I posted three new GotWs numbered #103-105. I decided leaving a gap in the numbers wasn't best after all, so I am renumbering them to #89-91 to continue the sequence. Here is the updated version of what was GotW #105.*

How should you prefer to pass smart pointers, and why?

Problem

JG Question

1. What are the *performance* implications of the following function declaration? Explain.

```
void f( shared_ptr<widget> );
```

Guru Questions

2. What are the *correctness* implications of the function declaration in #1? Explain with clear examples.

3. A colleague is writing a function f that takes an existing object of type widget as a required input-only parameter, and trying to decide among the following basic ways to take the parameter (omitting const):

```
 void f( widget* );              (a)
void f( widget& );              (b)
void f( unique_ptr<widget> );   (c)
void f( unique_ptr<widget>& );  (d)
void f( shared_ptr<widget> );   (e)
void f( shared_ptr<widget>& );  (f)
```

Under what circumstances is each appropriate? Explain your answer, including where const should or should not be added anywhere in the parameter type.

(There are other ways to pass the parameter, but we will consider only the ones shown above.)

Solution

1. What are the *performance* implications of the following function declaration? Explain.

```
void f( shared_ptr<widget> );
```

A shared_ptr stores strong and weak reference counts (see GotW #89). When you pass by value, you have to copy the argument (usually) on entry to the function, and then destroy it (always) on function exit. Let's dig into what this means.

When you enter the function, the shared_ptr is copy-constructed, and this requires incrementing the strong reference count. (Yes, if the caller passes a temporary shared_ptr, you *move*-construct and so don't have to update the count. But: (a) it's quite rare to get a temporary shared_ptr in normal code, other than taking one function's return value and immediately passing that to a second function; and (b) besides as we'll see most of the expense is on the destruction of the parameter anyway.)

When exiting the function, the shared_ptr is destroyed, and this requires decrementing its internal reference count.

What's so bad about a "shared reference count increment and decrement?" Two things, one related to the "shared reference count" and one related to the "increment and decrement." It's good to be aware of how this can incur performance costs for two reasons: one major and common, and one less likely in well-designed code and so probably more minor.

First, the major reason is the performance cost of the "increment and decrement": Because the reference count is an atomic shared variable (or equivalent), incrementing and decrementing it are internally-synchronized read-modify-write shared memory operations.

Second, the less-likely minor reason is the potentially scalability-bustingly contentious nature of the "shared reference count": Both increment and decrement update the reference count, which means that at the processor and memory level only one core at a time can be executing such an instruction on the same reference count because it needs exclusive access to the count's cache line. The net result is that this causes some contention on the count's cache line, which can affect scalability if it's a popular cache line being touched by multiple threads in tight loops—such as if two threads are calling functions like this one in tight loops and accessing shared_ptrs that own the same object. "So don't do that, thou heretic caller!" we might righteously say. Well and good, but the caller doesn't always know when two shared_ptrs used on two different threads refer to the same object, so let's not be quick to pile the wood around his stake just yet.

As we will see, an essential best practice for any reference-counted smart pointer type is to *avoid copying it unless you really mean to add a new reference*. This cannot be stressed enough. This directly addresses both of these costs and pushes their performance impact down into the noise for most applications, and especially eliminates the second cost because it is an antipattern to add and remove references in tight loops.

At this point, we will be tempted to solve the problem by passing the shared_ptr by reference. But is that really the right thing to do? It depends.

2. What are the *correctness* implications of the function declaration in #1?

The only correctness implication is that the function advertises in a clear type-enforced way that it will (or could) retain a copy of the shared_ptr.

That this is the only correctness implication might surprise some people, because there would seem to be one other major correctness benefit to taking a copy of the argument, namely lifetime: Assuming the pointer is not already null, taking a copy of the shared_ptr guarantees that the function itself holds a strong refcount on the owned object, and that therefore the object will remain alive for the duration of the function body, or until the function itself chooses to modify its parameter.

However, we already get this for free—thanks to structured lifetimes, the called function's lifetime is a strict subset of the calling function's call expression. Even if we passed the shared_ptr by reference, our function would as good as hold a strong refcount because *the caller already has one*—he passed us the shared_ptr in the first place, and won't release it until we return. (Note this assumes the pointer is not aliased. You have to be careful if the smart pointer parameter could be aliased, but in this respect it's no different than any other aliased object.)

> ***Guideline:*** *Don't pass a smart pointer as a function parameter unless you want to use or manipulate the smart pointer itself, such as to share or transfer ownership.*

> ***Guideline:*** *Prefer passing objects by value, *, or &, not by smart pointer.*

If you're saying, "hey, aren't raw pointers evil?", that's excellent, because we'll address that next.

3. A colleague is writing a function f that takes an existing object of type widget as a required input-only parameter, and trying to decide among the following basic ways to take the parameter (omitting const). Under what circumstances is each appropriate? Explain your answer, including where const should or should not be added anywhere in the parameter type.

(a) and (b): Prefer passing parameters by * or &.

```
void f( widget* );              (a)
void f( widget& );              (b)
```

These are the preferred way to pass normal object parameters, because they stay agnostic of whatever lifetime policy the caller happens to be using.

Non-owning raw * pointers and & references are okay to observe an object whose lifetime we know exceeds that of the pointer or reference, which is usually true for function parameters. Thanks to structured lifetimes, by default arguments passed to f in the caller outlive f's function call lifetime, which is extremely useful (not to mention efficient) and makes non-owning * and & appropriate for parameters.

Pass by * or & to accept a widget independently of how the caller is managing its lifetime. Most of the time, we don't want to commit to a lifetime policy in the parameter type, such as requiring the object be held by a specific smart pointer, because this is usually needlessly restrictive. As usual, use a * if you need to express null (no widget), otherwise prefer to use a &; and if the object is input-only, write const widget* or const widget&.

(c) Passing unique_ptr by value means "sink."

```
void f( unique_ptr<widget> );   (c)
```

This is the preferred way to express a widget-consuming function, also known as a "sink."

Passing a unique_ptr by value is only possible by moving the object and its unique ownership from the caller to the callee. Any function like (c) takes ownership of the object away from the caller, and either destroys it or moves it onward to somewhere else.

Note that, unlike some of the other options below, this use of a by-value unique_ptr parameter actually doesn't limit the kind of object that can be passed to those managed by a unique_ptr. Why not? Because any pointer can be explicitly converted to a unique_ptr. If we didn't use a unique_ptr here we would still have to express "sink" semantics, just in a more brittle way such as by accepting a raw *owning* pointer (anathema!) and documenting the semantics in comments. Using (c) is vastly superior because it documents the semantics in code, and requires the caller to explicitly move ownership.

Consider the major alternative:

```
// Smelly 20th-century alternative
void bad_sink( widget* p );  // will destroy p; PLEASE READ THIS COMMENT

// Sweet self-documenting self-enforcing modern version (c)
void good_sink( unique_ptr<widget> p );
```

And how much better (c) is:

```
// Older calling code that calls the new good_sink is safer, because
// it's clearer in the calling code that ownership transfer is going on
// (this older code has an owning * which we shouldn't do in new code)
//
widget* pw = ... ;

bad_sink ( pw );             // compiles: remember not to use pw again!

good_sink( pw );             // error: good
good_sink( unique_ptr<widget>{pw} );  // need explicit conversion: good

// Modern calling code that calls good_sink is safer, and cleaner too
//
unique_ptr<widget> pw = ... ;

bad_sink ( pw.get() );       // compiles: icky! doesn't reset pw
bad_sink ( pw.release() );   // compiles: must remember to use this way

good_sink( pw );             // error: good!
good_sink( move(pw) );       // compiles: crystal clear what's going on
```

Guideline: *Express a "sink" function using a by-value* ***unique_ptr*** *parameter.*

Because the callee will now own the object, usually there should be no const on the parameter because the const should be irrelevant.

(d) Passing unique_ptr by reference is for in/out unique_ptr parameters.

```
void f( unique_ptr<widget>& );  (d)
```

This should only be used to accept an in/out unique_ptr, when the function is supposed to actually accept an existing unique_ptr and potentially modify it to refer to a different object. It is a bad way to just accept a widget, because it is restricted to a particular lifetime strategy in the caller.

> ***Guideline:*** *Use a non-const* ***unique_ptr&*** *parameter only to modify the* ***unique_ptr****.*

Passing a const unique_ptr<widget>& is strange because it can accept only either null or a widget whose lifetime happens to be managed in the calling code via a unique_ptr, and the callee generally shouldn't care about the caller's lifetime management choice. Passing widget* covers a strict superset of these cases and can accept "null or a widget" regardless of the lifetime policy the caller happens to be using.

> ***Guideline:*** *Don't use a* ***const unique_ptr&*** *as a parameter; use* ***widget****** *instead.*

I mention widget* because that doesn't change the (nullable) semantics; if you're being tempted to pass const shared_ptr<widget>&, what you really meant was widget* which expresses the same information. If you additionally know it can't be null, though, of course use widget&.

(e) Passing shared_ptr by value implies taking shared ownership.

```
void f( shared_ptr<widget> );   (e)
```

As we saw in #2, this is recommended only when the function wants to retain a copy of the shared_ptr and share ownership. In that case, a copy is needed anyway so the copying cost is fine. If the local scope is not the final destination, just std::move the shared_ptr onward to wherever it needs to go.

> ***Guideline:*** *Express that a function will store and share ownership of a heap object using a by-value* ***shared_ptr*** *parameter.*

Otherwise, prefer passing a * or & (possibly to const) instead, since that doesn't restrict the function to only objects that happen to be owned by shared_ptrs.

(f) Passing shared_ptr& is useful for in/out shared_ptr manipulation.

```
void f( shared_ptr<widget>& );  (f)
```

Similarly to (d), this should mainly be used to accept an in/out shared_ptr, when the function is supposed to actually modify the shared_ptr itself. It's usually a bad way to accept a widget, because it is restricted to a particular lifetime strategy in the caller.

Note that per (e) we pass a shared_ptr by value if the function will share ownership. In the special case where the function *might* share ownership, but doesn't necessarily take a copy of its parameter on a given call, then pass a const shared_ptr& to avoid the copy on the calls that don't need it, and take a copy of the parameter if and when needed.

> ***Guideline:*** *Use a non-const* ***shared_ptr&*** *parameter only to modify the* ***shared_ptr****. Use a* ***const shared_ptr&*** *as a parameter only if you're not sure whether or not you'll take a copy and share ownership; otherwise use* ***widget****** *instead (or if not nullable, a* ***widget&****).*

Acknowledgments

Thanks in particular to the following for their feedback to improve this article: mttpd, zahirtezcan, Jon, GregM, Andrei Alexandrescu.

Published by Herb Sutter

Herb Sutter is an author and speaker, a software architect at Microsoft, and chair of the ISO C++ standards committee. View all posts by Herb Sutter

57 thoughts on "GotW #91 Solution: Smart Pointer Parameters"

Pingback: Fixed: C++ – passing references to std::shared_ptr or boost::shared_ptr #it #answer #dev | SevenNet

tohava says:
2014-12-15 at 11:59 am

1

1

i
Rate This

It is also useful to pass const shared_ptr & if the function creates a weak pointer from the shared pointer.

mikef says:
2014-09-01 at 8:35 pm

21

5

i
Rate This

Why are we making pointers even more complicated to teach and learn? Somehow we have figured out a way to make this more difficult to properly teach in a consistent manner and this makes learning exponentially harder.

Maybe it's time for a fundamental change. Let's not think about pointers, references, and smart pointers. Let's think about what we are really trying to do in these scenarios and bake the solutions into the language using different, more meaningful, names for those scenarios.

I know, I know… what about legacy code. Figure it out. This is supposed to be a means to an end. I have better things to do. Don't you?

Greg Marr says:
2014-08-22 at 1:06 pm

0

0

i
Rate This

Matt, there's a discussion on this exact topic going on right now over at Scott Meyer's blog, and Herb's involved.
http://scottmeyers.blogspot.com/2014/07/should-move-only-types-ever-be-passed.html

Matt B says:
2014-08-22 at 6:51 am

4

0

i

Rate This

Would defining good_sync() in your example as:

```
1  void good_sink( unique_ptr<widget>&& p );
```

prevent an extra std::unique_ptr from being generated (upon entry of the method), while enforcing the same semantics as the pass-by-value method declaration? The std::move() would do nothing to the std::unique_ptr passed into the function until the function itself actually “sunk” :) it, where as pass-by-value moves it immediately into a temporary.

What do you think about that approach?

kerjasambilandirumah.net says:
2014-05-09 at 5:18 am

0

2

i
Rate This

It’s an awesome paragraph in favor of all the online people; they will get advantage from it I am sure.

Matt Wilson says:
2014-02-06 at 10:53 am

12

0

i
Rate This

Please extend this article with the following two examples explained:
void f(weak_ptr); (e)
void f(weak_ptr&); (f)

I know some engineers that believe this is the best way to pass widget to methods that don’t take ownership.

Bartosz Milewski says:
2014-01-06 at 9:36 am

10

0

i
Rate This

If I saw (d) or (f) in a code review I would most likely send it back to the author for a rewrite with this note:

If these are just out parameters, return them:

```
1  unique_ptr<widget> f();  (d)
2  shared_ptr<widget> f();  (f)
```

If you need to return more than the widget, repack the other return values as out parameters.

If the widgets are in/out, you're really passing one widget as an in parameter and returning another, so separate them:

```
1  unique_ptr<widget> f(widget const * src_widget);  (d)
2  shared_ptr<widget> f(widget const * src_widget);  (f)
```

Using in/out parameters is already confusing enough; combining in and out ownership policies into one parameter is inviting a disaster.

Bret Kuhns says:
2013-12-19 at 9:56 am

2

0

i
Rate This

@CS There was a proposal for this for C++1y, but didn't make it into the draft. I believe the reasoning was that in the presence of `std::shared_ptr` and `std::unique_ptr`, raw points should always be expressed as observing. Unfortunately, your example of legacy code is exactly the reason why this was a bad decision. I would love to see the "worlds dumbest smart pointer" get revisited in the general library TS coming after C++14…

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3514.pdf

CS says:
2013-12-11 at 6:11 pm

4

0

i
Rate This

I work a large project that has legacy going back to the '80s. Some of the code is Fortran, some C, but a large part is C++ and most of the new code is C++.
Some of the C++ is in an old style and some has been updated to a more modern style.

I come across a function with a pointer parameter:

```
1  void DoSomethingAwesome(double *pVal);
```

Is this a modern “Non-owning raw * pointer”? or is this a legacy “bad_sink” that should be updated?

If I’m lucky there is a comment somewhere that tells me, otherwise I need to search into the .cpp file and decide if this is a bad_sink and perhaps update it. (Of course pVal in this function could actually be used as an iterator with some fun old school pointer math going on.)

It would be very nice if the language had a feature that allowed functions to explicitly state “pass me a non-owning pointer”. As it is now all they can say is “trust me, do I look like I might delete your non-owning raw pointer?”

Pingback: C&B Background Reading, Part Deux | C++ and Beyond
Herb Sutter says:
2013-12-03 at 8:51 am

2

0

i
Rate This

Edited 2013-12-03 to implement private feedback from Andrei Alexandrescu: Rewrote parts of the solution to #1 to make a correction (remove the comment about increment being cheap, as there overhead and it does incur a fence on x86/x64) and add the following new explanatory paragraph to drive home an essential point made again later on:

New text: “As we will see, an essential best practice for any reference-counted smart pointer type is to avoid copying it unless you really mean to add a new reference. This cannot be stressed enough. This directly addresses both of these costs and pushes their performance impact down into the noise for most applications, and especially eliminates the second cost because it is an antipattern to add and remove references in tight loops.”

Thanks, Andrei!

leo_carvalho says:
2013-11-19 at 4:02 pm

0

0

i
Rate This

@jlehrer
You can use template alias for that.

```
1 template <class T>
2 using pointer = T*;
```

earwicker says:
2013-07-06 at 2:13 am

0

0

i
Rate This

GregM, i that's what you took away from my last comment, then obviously I worded it extraordinarily badly. I apologise! Thanks for the effort anyway.

In case anyone else might still be interested, I have an issue with the advice offered in "2. What are the correctness implications…"

– Object is being managed by a shared_ptr
– Advice: we should pass a raw pointer to that object to some function
– Caveat: "Note this assumes the pointer is not aliased"
– The raw pointer we passed to the function is itself an alias of the original pointer.

So we can follow this advice as long as we don't follow this advice?

GregM says:
2013-07-05 at 5:04 pm

0

0

i
Rate This

Given that what you want is exactly what you get here, then we all agree. Thanks for your time. Goodbye.

earwicker says:
2013-07-05 at 2:20 pm

0

0

i
Rate This

And again, refer to Herb's section that spans "2. What are the correctness implications… but in this respect it's no different than any other aliased object.)"

He is not talking at all about the internal state of the pointed-to object at all. He's talking only about "lifetime", how it seems that "taking a copy of the shared_ptr guarantees that the function itself holds a strong refcount on the owned object, and that therefore the object will remain alive for the duration of the function body, or until the function itself chooses to modify its parameter."

But he goes on: "However, we already get this for free… (except we don't)" I elided/paraphrased that last part, but quite adequately.

Hence just as there's no need to mention threads, there's also no need to suppose some mutable state of the pointed-to object. The entire point here is purely and only about the lifetime of the object, as controlled by the one-or-more shared_ptrs that point to it.

To make this absolutely concrete: what if the pointed-to object is immutable? Say we read a big map-of-strings-to-strings from a JSON file, and then encapsulate it inside a Config class that only allows read access to the map. This means that once loaded, we can safely share our single Config instance around (between threads or otherwise) and know its contents won't get mutated. But we want it to be deleted when no longer in use. It's a perfect application for shared_ptr. Even though the Config is visible from multiple places, no one can modify it, and so the only additional guarantee we need is that it won't get destroyed at the wrong time.

GregM says:
2013-07-04 at 10:28 am

0

0

i
Rate This

Okay, fine, so leave threads out of it. If you call some other function that does something to the object that you're working on, then it's still a problem. This same problem exists whether you have a copy of the shared_ptr or a reference to the shared_ptr. The copy of the shared_ptr only gives you additional safety against the object being destroyed if there is someone else that has a non-const reference to the shared_ptr itself. In that case, however, it still gives you no safety against changes in the object itself.

If you and your caller both have a pointer to the object, but so does someone else, and you call that someone else to do something, and it, unknown to you, changes all the data in that object, how has the copy of the shared_ptr helped you?

Analogies don't generally work very well, but let me give it a shot with what you used.

Say that you are holding a single red rose between your teeth, and I'm holding your hand. The other person who is also holding the same single red rose between their teeth can bite through the stem and ruin the rose. My holding the rose between my teeth in addition to holding your hand doesn't prevent the other person from biting through the stem. If you want the other person to not be able to bite through the stem, you better make sure that they don't have it between their teeth. It won't make a bit of difference how I hold the rose.

Since you're holding the rose between your teeth, and you are frozen until I go away, then I know that you can't open your mouth and drop the rose. I can either simply hold your hand, or I can hold your hand and also hold the rose between my teeth, and the rose won't be dropped on the ground no matter what happens.

earwicker says:
2013-07-04 at 2:18 am

0

0

i
Rate This

@GregM – re: “data races with your smart pointer” – this is about shared_ptr, not any old smart pointer. The standard shared_ptr is in fact thread-safe w.r.t. assignment. This is why it is so slow, which in turn is why Herb is so keen on discouraging patterns where shared_ptr gets copied a lot.

And in any case, re: the part I’m questioning, why bring threads into it? Herb (correctly) does not..His “2. What are the correctness implications…” makes no mention at all of data races or multiple threads, and nor does it need to: it’s about the relationship between variables declared at different places in a single call/variable stack, such as is found in a single-threaded C++ program.

Herb: “Even if we passed the shared_ptr by reference, our function would as good as hold a strong refcount because the caller already has one—he passed us the shared_ptr in the first place, and won’t release it until we return. (Note this assumes the pointer is not aliased… in this respect it’s no different than any other aliased object.)”

That guarantee (including the disclaimer-in-brackets that sadly undermines it) is all about a single thread.

My point is that it’s about as helpful as:

Me: “If I hold a single red rose between my teeth, you and I can tango into the Room of Poison Gas without dying! (Assuming of course that there’s no poison gas in the room at this particular moment – but hey, poison gas is generally dangerous so why should this little tacked-on caveat be any kind of surprise?)”

Emmanuel Thivierge says:
2013-07-02 at 5:27 pm

0

0

i
Rate This

Hi everybody,
I came back o read again this post because I was in front of a dilema and I was wondering what is the best way to express it.

I had created a std:vector and I was going to pass it to a function

std:vector widgetList;
myWidgetManager.addWidgetList(widgetList);
[\code]

To me addWidgetList is a sink, the object will take the ownership of the vector. So I tough, I should pass a std:unique_ptr<std::vector>, first it start to be a mouthful to write and my object is actually on the stack.

In this guru you don’t only talk about smart pointers but also on when/how to pass argument to functions. To me this situation is like c) the function is a sink but for the stack i would express it as std::vector&&.

Would that be a good practice?
thanks
Mani

GregM says:
2013-07-01 at 8:05 am

0

0

i
Rate This

The point is that if you have data races with your smart pointer or your pointed-to object, then neither by-copy or by-ref is going to help you.

earwicker says:
2013-06-25 at 4:06 am

0

0

i
Rate This

@GregM

“which is a problem no matter which kind of pointer”

See, Herb made exactly the same kind of observation in his parenthetical caveat: “… but in this respect it’s no different than any other aliased object.”

What difference does it make to whether we need to take care in this situation?

It’s not just a problem for pointers, it’s a problem for any object with mutable internal state, e.g. a string vs. a ‘const string&’

This doesn’t stop it from being a problem in the case of shared_ptr, does it?

By accepting a parameter that is a const &, you’ve already introduced at least one alias. You’ve said, in effect, “I don’t care if this object stays alive – I assume all my callers will ensure that for me”.

Rather than saying “it’s a guarantee (except it’s not)”, why not instead say we always have the same trade-off to make:

1. copy (slow, but ensures object will survive)
2. ref (fast, doesn’t ensure object will survive)

If you follow 2 by default throughout your code, you are practicing the art of “write fast code, then fix it”. If you follow 1 by default, you’re following “write correct code, then optimize it”.

Again, choose your poison! Due to the severe performance problem caused by all the ref-counting, optimizing a correct program may be as hard as finding/fixing bugs in a fast program.

GregM says:
2013-06-21 at 11:20 am

0

0

i
Rate This

"What actual firm claim is being made here?"

Absent aliasing, which is a problem no matter which kind of pointer you are using, and can't be dealt with just by using a different kind of pointer, you can be guaranteed that the object will be alive for as long as the function call.

"When can it be relied on? How can we tell?"

It can be relied on as long as there is no aliasing of the pointer. As for how you can tell, that's going to require that you look at your program to see if there is any aliasing going on.

earwicker says:
2013-06-21 at 5:27 am

1

0

i
Rate This

re: Herb's caveat "(Yes, you have to be careful if the smart pointer parameter can be aliased, but in this respect it's no different than any other aliased object)."

Reminds me of the guy who was hitting himself with a hammer. When his doctor told him "Stop hitting yourself with a hammer", he replied "Oh, come on, you can't blame the hammer. It would be just as bad if I hit myself with a brick or anything else for that matter!"

Yes, if g calls f then as long as f is still running, then g is still running. But g could pass to f a member variable, m, which presently refers to an object, and then some unholy, filthy chain of callbacks set off within f could reach back and cause m to be mutated, all while f – and therefore g – still have not returned.

So Herb's paragraph "However, we already get this for free…" would seem to be followed by the caveat "… (except when we don't)." What actual firm claim is being made here? When can it be relied on? How can we tell?

Pingback: Herb Sutter GotW91: Smart Pointer parameters | musingstudio
Mathias Stearn says:
2013-06-08 at 11:40 am

2

0

i
Rate This

What do you think of using something like ptr [1] which is usable like a T* but is implicitly convertible from any smart pointer and clearly marks non-owning semantics? I've found that in code that uses smart pointers, functions that take a bare pointer are annoying because you have to call get(), breaking the abstraction. It also has a nice side benefit of less visual ambiguity between const ptr and ptr.

On the related issue of pointer returns, do you know why covariant return support wasn't added for smart pointers? It seems like a clone() method would have to choose between better type-safety but using a raw owning pointer or returning a smart pointer and requiring casts.

[1] https://gist.github.com/RedBeard0531/5736292 (written for C++98 + boost)

Herb Sutter says:
2013-06-07 at 5:03 am

0

0

i
Rate This

@Jon: That's probably a rare case but I think you're right it's worth calling out. Added, thanks.

Jon says:
2013-06-06 at 5:26 pm

2

0

i
Rate This

@Herb: If a function is going to decide at runtime whether to share ownership or not, should we still pass the shared_ptr by value or is const reference a legitimate choice? It seems to me that if we pass an lvalue by value and the called function decides not to share ownership, we've copied a share_ptr for no reason.

Brian Fiete says:
2013-06-06 at 6:32 am

1

0

i

Rate This

Re: #2 (correctness) – even without changing the state of the calling function, passing a shared_ptr by reference can be hazardous if the function is able to remove the shared_ptr reference that was passed to it. A trivial case is below (results in a crash), but one can easily imagine a less trivial case — say, if the function were to run some validation on the data passed to it which could result in a new instantiation of that data…

```
1  struct IntArrayContainer
2  {
3      shared_ptr<vector<int>> mIntArray;
4  };
5
6  IntArrayContainer* gIntArrayContainer;
7
8  int GetFirstValueFrom(shared_ptr<vector<int>>& intArray)
9  {
10     delete gIntArrayContainer;
11     return (*intArray)[0];
12 }
13
14 void main()
15 {
16     gIntArrayContainer = new IntArrayContainer();
17     gIntArrayContainer->mIntArray = shared_ptr<vector<int>>(new vector<i
18     printf("First: %d\n", GetFirstValueFrom(gIntArrayContainer->mIntArra
19 }
```

Kamil Rojewski says:
2013-06-06 at 1:53 am

1

0

i
Rate This

I have a question about example (c). Wouldn't it be better to use rvalue ref to unique_ptr instead of pass-by-value? It still requires to pass an explicit unique_ptr, but it shows an expcilit move is done, instead of implicit by passing to the parmateter value.

Leszek Swirski says:
2013-06-06 at 1:28 am

0

0

i
Rate This

@Jon: I have considered it, but a similar proposal was rejected in 2003, because (as I understand it), even people who liked the syntax didn't like the fact that this was essentially optional, rather than a language change (cf. Ada, C#). The argument was that for

```
1 f(out(x),y)
```

you technically still don't know if y is mutable or not, because the evil library writer could still be using normal references.

http://lists.boost.org/Archives/boost/2003/04/47180.php

decourse says:
2013-06-05 at 11:34 pm

1

0

i
Rate This

As I noted in a comment on the question, there is another use for (d). Some functions may choose to either take or not take ownership of an object by criteria to be decided at run-time, perhaps by inspecting the object itself. Think of a "chain of responsibility" pattern, for example.

Herb Sutter says:
2013-06-05 at 9:15 pm

0

0

i
Rate This

@Jon: This is better in the case where you do get an rvalue, and the default guidance to "pass by value then move as needed" doesn't apply just to smart pointers but to any type that's cheaper to move than copy. And yes, my opinion has changed. Basically that came from paranoia about a different optimization that doesn't apply to standard C++ code (something we encountered designing C++/CX that can apply to the underlying implementation of refcounted ^'s but I now realize doesn't apply to shared_ptrs in normal C++ code).

jlehrer says:
2013-06-05 at 6:16 pm

3

0

i
Rate This

Re: "A Proposal for the World's Dumbest Smart Pointer"

I've always wished the C++ language supported a simpler alternative syntax for pointers:

```
pointer<int> <--> int*
pointer<const int> <--> const int *
const pointer<int> <--> int * const
const pointer<const int> <--> const int * const
```

this would make teaching pointer syntax to new developers much easier.

Obviously, we can add this as a language feature using

```
template <typename T> struct pointer {
 typedef T* type;
};
pointer<int>::type
```

But this is not the same as a language feature. For example, the compiler would not be able to compile the following code due to the templated function parameter:

```
template <typename T> void func(typename pointer<T>::type);
int x;
func(&x); //fails to compile
```

Jon says:
2013-06-05 at 6:00 pm

0

0

i
Rate This

@Leszek Swirski: Have you considered submitting your header to boost?

Having mutability visible at the call site is useful.
Eg Bjarne Stroustrup: http://www.stroustrup.com/bs_faq2.html#call-by-reference
“My personal style is to use a pointer when I want to modify an object because in some contexts that makes it easier to spot that a modification is possible.”
@Herb, would it be fair to say you disagree with the above style? :)

Jon says:
2013-06-05 at 5:37 pm

1

0

i
Rate This

@Herb: “If the local scope is not the final destination, just std::move the shared_ptr onward to wherever it needs to go.” Why is this better than passing by const shared_ptr& through a chain of functions until we’re ready for the final destination? What if the callee will make a decision about whether to share ownership or not? If it decides not to, then we’ve copied for no reason.

Also I noted that you said “always pass them by reference to const, and very occasionally maybe because you know what you called might modify the thing you got a reference from, maybe then you might pass by value” (http://stackoverflow.com/a/8844924/297451). Can I confirm that your opinion has changed since then?

Bret Kuhns says:
2013-06-05 at 4:49 pm

3

0

i
Rate This

@Herb, since reference types for `std::optional` was not approved, it makes it all the more unfortunate that N3514 “A Proposal for the World’s Dumbest Smart Pointer” didn’t make it either. A `std::exempt_ptr` would be an obvious answer to explicitly express non-owning pointer semantics.

Herb Sutter says:
2013-06-05 at 4:39 pm

0

0

i
Rate This

@mttpd: I’m using WordPress.com, not a WP.org installation. In return for much less admin hassle, it’s more restrictive — I can’t install packages, only use what they give me. I’ll poke around and see if there’s something available though.

Herb Sutter says:
2013-06-05 at 4:38 pm

2

0

i
Rate This

@BrianM: But the Guidelines are simple — don’t pass by smart pointer unless you want to use/modify the smart pointer itself (like any object). The main thing is that pass by value/*/& are all still good and should still be used primarily. It’s just that we now have a couple of idioms for expressing ownership transfer in function signatures, notably passing a unique_ptr by value means “sink” and passing a shared_ptr by value means “gonna share ownership.” That’s pretty much it.

Leszek Swirski says:
2013-06-05 at 4:27 pm

1

0

i
Rate This

For mutable parameters, I like to use my tiny inout header[1], which provides out and inout parameters in the same style as C#'s out[2] and ref[3]. That way the mutability is visible at the call site, there are no raw pointers in user code, and there is no question as to whether the referenced variable needs to be initialised or not (because it only has to be initialised for an inout variable).

[1] https://gist.github.com/LeszekSwirski/3028820
[2] http://msdn.microsoft.com/en-us/library/ee332485.aspx
[3] http://msdn.microsoft.com/en-us/library/14akc2c7.aspx

Brian M says:
2013-06-05 at 3:50 pm

10

1

i
Rate This

What worries me here is the number of ways and opinions going on – with real world coders I just wonder what would be used in the average non-guru shop or guru populated shop for that matter! The use of the raw pointer does have its simplicity in this case and the only really good argument against it is in a sink function, even then a suitable name such as ProcessAndDelete might suffice. If a coder can't pick up on that, then we are probably doomed anyway!

Beginning to think we are now in no-mans land with c++, it's getting way too complicated, after all its only a tool to achieve an end. Maybe we need a more intelligent compiler that can make a lot of these 'guru' decisions for the average coder. It should not beyond the ability of a compiler to have a autoptr mk2 that is compiled into the most efficient type for the job. Be like having Herb or Meyers standing behind you :)

mttpd says:
2013-06-05 at 3:27 pm

0

0

i
Rate This

@Herb: fair enough. I can see you're using WordPress, how about adding WP-Markdown to enable MarkDown (also mentioned by Bret, it's what's used on GitHub, Reddit, StackOverflow, and many others — and supports the backticks syntax for the inline code formatting): http://wordpress.org/plugins/wp-markdown/

// Here's the relevant feature: http://daringfireball.net/projects/markdown/syntax#code

Bret Kuhns says:
2013-06-05 at 2:42 pm

1

1

i
Rate This

@Herb, Well that's unfortunate; I assumed std::optional would work like boost's (plus move semantics). I can see the point that reference types represent a somewhat awkward subset of std::optional T.

Also, I assume your own comment got mangled when you said "std::optional isn't standrad" ;-)

Herb Sutter says:
2013-06-05 at 2:30 pm

2

1

i
Rate This

@mttpd: Maybe I should allude to this, thanks: Because const shared_ptr& and widget* have the same (nullable) semantics. If you're being tempted to pass const shared&, widget* doesn't lose information, whereas switching to widget& can lose information. Sure, use a & if it can't be null. And re code: All I know of is for code blocks.

@David: Normally you're right to be very suspicious about claims of no barriers/synchronization. In this case, the key is that because we can guarantee that no action is never taken by another thread because of a refcount increment, it is not really a "publish" operation, so one can apply subtle reasoning and prove that the increment can be memory_order_relaxed; but it's exceedingly subtle and most experts shouldn't even try to venture into waters like these. I explain this example toward the end of my "atomic Weapons, Part 2" talk. See pages 50-52 of the slides at http://sdrv.ms/NxDB6u . The talk video is at http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2 .

@zahirtezcan, Francis: Once you've copied the shared_ptr, If you want to move the shared_ptr onward, that's fine, just std::move it. Constructors aren't special, this applies to any function. I should mention this to avoid confusion; added, thanks.

@Bret: Alas, std::optional isn't standard; see that part of Andrzej's paper at http://isocpp.org/files/papers/N3672.html#optional_ref . I asked Andrzej about it a few days ago and that part wasn't accepted for C++14 and isn't likely to be. There is a workaround: std::optional<reference_wrapper>. IMO it's less clear, less efficient, and more verbose than plain widget*.

@Bret,Arthur: Yes, I intend to eventually write a GotW on parameter passing. Probably not for a couple of months though, if I do get to it. And I may not.

Bret Kuhns says:
2013-06-05 at 1:48 pm

0

0

i
Rate This

@ Róbert Dávid,

```
1 std::optional<Widget&>
```

would “own” the reference, not the `Widget` being referenced to. I already use

```
1 boost::optional<Widget&>
```

in code to express “non-owning observing reference”.

Róbert Dávid says:
2013-06-05 at 1:06 pm

2

0

i
Rate This

BTW, it’s better if the function in 3(a) is a pointer constant, not (just) a pointer-to-constant:

f(widget * const);
f(const widget * const);

Doesn’t matter too much (as it’s just a matter of a copy to get a modifiable pointer), but it does document that the function will inspect only the pointed object, and won’t do tricky traversals.

Róbert Dávid says:
2013-06-05 at 1:01 pm

0

0

i
Rate This

@Francis Rammeloo: If he moves w into the container, then no..

Róbert Dávid says:
2013-06-05 at 1:01 pm

1

0

i
Rate This

@Bret Kuhns: Raw pointers in modern C++ are not good (read: should not be used) for anything else than that!

The problem with optional that it owns the object. The exact idea behind point 3a/b is that the function does not try to manage the lifetime. There you need to have a non-owning reference: You can have reference semantics, use &, or if it can be nullable, use a raw pointer.

Francis Rammeloo says:
2013-06-05 at 12:14 pm

0

0

i
Rate This

Concerning case e:

```
1  void f( shared_ptr<widget> w);
```

The receiver will likely store the copy in a container. Will this incur the cost of a second copy?

zenmumbler says:
2013-06-05 at 9:51 am

0

0

i
Rate This

That'll teach me from not refreshing the page before I post a comment. Good thing I'm not the only one thinking happy thoughts about optional.

Arthur Langereis says:
2013-06-05 at 9:49 am

2

1

i
Rate This

Also, given that we're talking C++14 here:

when you indicate that widget* should be used for optional objects, why not advise to use std::optional as a parameter instead? In fact, a GotW on optional would be appreciated as I'm sure there are subtleties to it that require a closer look.

bkuhns says:
2013-06-05 at 8:55 am

3

0

i
Rate This

The template parameters were edited out of my comment above. Should say

```
1 std::optional<Widget&>
```

every time I say "std::optional".

Herb, pretty please, find a way to support markdown in comments on your blog!

Bret Kuhns says:
2013-06-05 at 8:31 am

12

1

i
Rate This

Are non-owning raw pointers really such a worth-while idea in C++14? What are they good for anymore in "modern" code? If I want to observe something that can optionally be null, wouldn't std::optional be a better candidate than Widget*?

Additionally, in the real world, not all code is necessarily modern. So although all raw pointers should be only observing in modern code, legacy code still uses them to mean something else. Now you have to ask yourself: is this function that wants a Widget* modern, or is it still living by old implicit ownership semantics with raw pointers? If the function is living by modern guidelines, it's not doing a good job of telling you that in its signature. Using std::optional eliminates this ambiguity.

zahirtezcan (@zahirtezcan) says:
2013-06-05 at 7:58 am

1

1

i
Rate This

I failed to use segments…

zahirtezcan (@zahirtezcan) says:
2013-06-05 at 7:56 am

5

0

i
Rate This

For using “widget*” instead of “const shared_ptr&” guideline, what about constructors taking a shared_ptr? Is it supposed to use shared_ptr by value and move the incoming parameter to the member? Or take as a modifiable reference and make copy out of it?

```
struct SomeStruct
{
shared_ptr memberPtr;

SomeStruct(const shared_ptr& in): memberPtr(in){}// <– this one?
Somestruct(shared_ptr in): : memberPtr(std::move(in)){}// <– or, this one?
Somestruct(shared_ptr& in): memberPtr(in){}// <– or, this one?
};
```

David Oster says:
2013-06-05 at 7:52 am

13

1

i
Rate This

Section 1 of this says that incrementing the reference count of a shared_ptr can be optimized to a single read-modify-write without guards, while a decrement must have guards. I’d love to see a link to an explanation of this statement. I thought that if one writer needs a synchronization primitive then all writers needed that primitive.

mttpd says:
2013-06-05 at 7:03 am

5

1

i
Rate This

Regarding the following in the last guideline: “Don’t use a const shared_ptr& parameter; use widget* instead.”
Why not widget&?

BTW, is there a way to enable inline code formatting, e.g., using backticks ` ` like on StackOverflow?

Lars Viklund says:
2013-06-05 at 6:57 am

10

1

i
Rate This

The assumption in (2) that the caller will not destroy their shared_ptr that's passed by reference does not always hold. It's trivial that from the leaf function invoke something that indirectly changes the state of the calling function, not uncommon when there's callbacks and notifications involved. I would augment the advise to only recommend references to shared_ptr if you're properly sure that they are valid for the duration of the call.

Comments are closed.