How much null checking is enough?

Asked 16 years, 1 month ago Modified 10 years, 11 months ago Viewed 13k times

68

What are some guidelines for when it is **not** necessary to check for a null?

A lot of the inherited code I've been working on as of late has null-checks ad nauseam. Null checks on trivial functions, null checks on API calls that state non-null returns, etc. In some cases, the null-checks are reasonable, but in many places a null is not a reasonable expectation.

I've heard a number of arguments ranging from "You can't trust other code" to "ALWAYS program defensively" to "Until the language guarantees me a non-null value, I'm always gonna check." I certainly agree with many of those principles up to a point, but I've found excessive null-checking causes other problems that usually violate those tenets. Is the tenacious null checking really worth it?

Frequently, I've observed codes with excess null checking to actually be of poorer quality, not of higher quality. Much of the code seems to be so focused on null-checks that the developer has lost sight of other important qualities, such as readability, correctness, or exception handling. In particular, I see a lot of code ignore the std::bad_alloc exception, but do a null-check on a `new`.

In C++, I understand this to some extent due to the unpredictable behavior of dereferencing a null pointer; null dereference is handled more gracefully in Java, C#, Python, etc. Have I just seen poor-examples of vigilant null-checking or is there really something to this?

This question is intended to be language agnostic, though I am mainly interested in C++, Java, and C#.

Some examples of null-checking that I've seen that seem to be *excessive* include the following:

This example seems to be accounting for non-standard compilers as C++ spec says a failed new throws an exception. Unless you are explicitly supporting non-compliant compilers, does this make sense? Does this make *any* sense in a managed language like Java or C# (or even C++/CLR)?

```
try {
   MyObject* obj = new MyObject();
   if(obj!=NULL) {
      //do something
```

```
    } else {
        //??? most code I see has log-it and move on
        //or it repeats what's in the exception handler
    }
} catch(std::bad_alloc) {
    //Do something? normally--this code is wrong as it allocates
    //more memory and will likely fail, such as writing to a log file.
}
```

Another example is when working on internal code. Particularly, if it's a small team who can define their own development practices, this seems unnecessary. On some projects or legacy code, trusting documentation may not be reasonable... but for new code that you or your team controls, is this really necessary?

If a method, which you can see and can update (or can yell at the developer who is responsible) has a contract, is it still necessary to check for nulls?

```
//X is non-negative.
//Returns an object or throws exception.
MyObject* create(int x) {
    if(x<0) throw;
    return new MyObject();
}

try {
    MyObject* x = create(unknownVar);
    if(x!=null) {
        //is this null check really necessary?
    }
} catch {
    //do something
}
```

When developing a private or otherwise internal function, is it really necessary to explicitly handle a null when the contract calls for non-null values only? Why would a null-check be preferable to an assert?

(obviously, on your public API, null-checks are vital as it's considered impolite to yell at your users for incorrectly using the API)

```
//Internal use only--non-public, not part of public API
//input must be non-null.
//returns non-negative value, or -1 if failed
int ParseType(String input) {
    if(input==null) return -1;
    //do something magic
    return value;
}
```

Compared to:

```
//Internal use only--non-public, not part of public API
//input must be non-null.
//returns non-negative value
int ParseType(String input) {
   assert(input!=null : "Input must be non-null.");
   //do something magic
   return value;
}
```

c# java c++ design-by-contract

Share
Improve this question
Follow

edited Nov 19, 2008 at 17:48
Greg Hewgill
990k ● 191 ● 1.2k ● 1.3k

asked Nov 19, 2008 at 17:39
James Schek
18k ● 7 ● 53 ● 64

new will NEVER return NULL (Unless you explicitly requirest it with the alternative new). So there is not need to check after a new or from anything that can not return NULL (because internally it is throwing). – Loki Astari Nov 19, 2008 at 18:14

1 Older C++ compilers violated the spec in this regard and returned a NULL instead of throwing an exception. Microsoft was one of the bigger violators, but not the only one. support.microsoft.com/kb/167733 – James Schek Nov 19, 2008 at 18:22

1 I don't think assert() is appropriate in your 4th sample. A NullPointerException would be more appropriate. There are legitimate uses for assert() but this isn't one of them. – finnw Nov 20, 2008 at 14:58

Assertions are not checked in production code and cannot be used for this scenario. Unless you make your own `assert(boolean, String)` method and inherit it or `static import` it everywhere `;)` – ADTC Sep 9, 2014 at 15:42

18 Answers

Sorted by: Highest score (default)

38

One thing to remember that your code that you write today while it may be a small team and you can have good documentation, will turn into legacy code that someone else will have to maintain. I use the following rules:

1. If I'm writing a public API that will be exposed to others, then I will do null checks on all reference parameters.
2. If I'm writing an internal component to my application, I write null checks when I need to do something special when a null exists, or when I want to make it very clear. Otherwise I don't mind getting the null reference exception since that is also fairly clear what is going on.

3. When working with return data from other peoples frameworks, I only check for null when it is possible and valid to have a null returned. If their contract says it doesn't return nulls, I won't do the check.

Share
Improve this answer
Follow

edited Nov 20, 2008 at 14:56
Michael Haren
108k ● 41 ● 171 ● 206

answered Nov 19, 2008 at 17:53
JoshBerke
67k ● 25 ● 129 ● 170

20

First note that this a special case of contract-checking: you're writing code that does nothing other than validate at runtime that a documented contract is met. Failure means that some code somewhere is faulty.

I'm always slightly dubious about implementing special cases of a more generally useful concept. Contract checking is useful because it catches programming errors the first time they cross an API boundary. What's so special about nulls that means they're the only part of the contract you care to check? Still,

On the subject of input validation:

null is special in Java: a lot of Java APIs are written such that null is the only invalid value that it's even possible to pass into a given method call. In such cases a null check "fully validates" the input, so the full argument in favour of contract checking applies.

In C++, on the other hand, NULL is only one of nearly 2^32 (2^64 on newer architectures) invalid values that a pointer parameter could take, since almost all addresses are not of objects of the correct type. You can't "fully validate" your input unless you have a list somewhere of all objects of that type.

The question then becomes, is NULL a sufficiently common invalid input to get special treatment that `(foo *)(-1)` doesn't get?

Unlike Java, fields don't get auto-initialized to NULL, so a garbage uninitialized value is just as plausible as NULL. But sometimes C++ objects have pointer members which are explicitly NULL-inited, meaning "I don't have one yet". If your caller does this, then there is a significant class of programming errors which can be diagnosed by a NULL check. An exception may be easier for them to debug than a page fault in a library they don't have the source for. So if you don't mind the code bloat, it might be helpful. But it's your caller you should be thinking of, not yourself - this isn't defensive coding, because it only 'defends' against NULL, not against (foo *)(-1).

If NULL isn't a valid input, you could consider taking the parameter by reference rather than pointer, but a lot of coding styles disapprove of non-const reference parameters. And if the caller passes you *fooptr, where fooptr is NULL, then it has done nobody any good anyway. What you're trying to do is squeeze a bit more documentation into the function signature, in the hope that your caller is more likely to think "hmm, might fooptr be null here?" when they have to explicitly dereference it, than if they just pass it to you as a pointer. It only goes so far, but as far as it goes it might help.

I don't know C#, but I understand that it's like Java in that references are guaranteed to have valid values (in safe code, at least), but unlike Java in that not all types have a NULL value. So I'd guess that null checks there are rarely worth it: if you're in safe code then don't use a nullable type unless null is a valid input, and if you're in unsafe code then the same reasoning applies as in C++.

On the subject of output validation:

A similar issue arises: in Java you can "fully validate" the output by knowing its type, and that the value isn't null. In C++, you can't "fully validate" the output with a NULL check - for all you know the function returned a pointer to an object on its own stack which has just been unwound. But if NULL is a common invalid return due to the constructs typically used by the author of the callee code, then checking it will help.

In all cases:

Use assertions rather than "real code" to check contracts where possible - once your app is working, you probably don't want the code bloat of every callee checking all its inputs, and every caller checking its return values.

In the case of writing code which is portable to non-standard C++ implementations, then instead of the code in the question which checks for null and also catches the exception, I'd probably have a function like this:

```
template<typename T>
static inline void nullcheck(T *ptr) {
    #if PLATFORM_TRAITS_NEW_RETURNS_NULL
        if (ptr == NULL) throw std::bad_alloc();
    #endif
}
```

Then as one of the list of things you do when porting to a new system, you define PLATFORM_TRAITS_NEW_RETURNS_NULL (and maybe some other PLATFORM_TRAITS) correctly. Obviously you can write a header which does this for all the compilers you know about. If someone takes your code and compiles it on a non-standard C++ implementation that you know nothing about, they're fundamentally on their own for bigger reasons than this, so they'll have to do it themselves.

Share

Improve this answer

Follow

edited Jan 15, 2014 at 15:00

answered Nov 21, 2008 at 12:27

Steve Jessop

279k ● 40 ● 469 ● 709

10

If you write the code and its contract, you are responsible for using it in terms of its contract and ensuring the contract is correct. If you say "returns a non-null" x, then the caller should not check for null. If a null pointer exception then occurs with that reference/pointer, it is your contract that is incorrect.

Null checking should only go to the extreme when using a library that is untrusted, or does not have a proper contract. If it is your development team's code, stress that the contracts must not be broken, and track down the person who uses the contract incorrectly when bugs occur.

Share Improve this answer Follow

answered Nov 19, 2008 at 17:48

MetroidFan2002

29.9k ● 16 ● 65 ● 81

He's talking about when you didn't write the code or (presumably) the contract. Doesn't address the point. – dkretz Nov 19, 2008 at 18:10

I think he means "you" as in the generic developer, not me specifically... I think the second paragraph is relevant. I may not have written the code, but I can talk to the author in some cases. – James Schek Nov 19, 2008 at 18:25

Why not check a value returned from a function which says in the contract that it won't return a NULL? The contract is documentation, the function is code. They are not, in C++ at least, necessarily coherent. The code is not necessarily correct. You just pass the bug. Fail early and often. – n-alexander Nov 20, 2008 at 14:45

Because it will fail when the contract is incorrect, and you get a segmentation fault. So your null check will not help, and the contract should be updated if it is code under your control, or your team's control. If you trust yourself and your team, and have a good process, then code is clearer. – MetroidFan2002 Nov 20, 2008 at 14:57

Maybe the fundamental issue here is not really about contract vs code, but whether you can manage policy through technology. – James Schek Nov 20, 2008 at 15:54

7

Part of this depends on how the code is used -- if it is a method available only within a project vs. a public API, for example. API error checking requires something stronger than an assertion.

So while this is fine within a project where it's supported with unit tests and stuff like that:

```
internal void DoThis(Something thing)
{
    Debug.Assert(thing != null, "Arg [thing] cannot be null.");
    //...
}
```

in a method where you don't have control over who calls it, something like this may be better:

```
public void DoThis(Something thing)
{
    if (thing == null)
    {
        throw new ArgumentException("Arg [thing] cannot be null.");
    }
    //...
}
```

Share Improve this answer Follow

answered Nov 19, 2008 at 17:56

djuth

985 ● 1 ● 8 ● 9

2 Code appears to be a C# example so Something is likely a nullable-reference.
– James Schek Nov 19, 2008 at 18:27

Yep, it's C# so thing can certainly be null. – djuth Nov 20, 2008 at 18:53

7

It depends on the situation. The rest of my answer assumes C++.

- I never test the return value of new since all the implementations I use throw bad_alloc on failure. If I see a legacy test for new returning null in any code I'm working on, I cut it out and don't bother to replace it with anything.
- Unless small minded coding standards prohibit it, I assert documented preconditions. Broken code which violates a published contract needs to fail immediately and dramatically.
- If the null arises from a runtime failure which isn't due to broken code, I throw. fopen failure and malloc failure (though I rarely if ever use them in C++) would fall into this category.
- I don't attempt to recover from allocation failure. Bad_alloc gets caught in main().
- If the null test is for an object which is collaborator of my class, I rewrite the code to take it by reference.
- If the collaborator really might not exist, I use the Null Object design pattern to create a placeholder to fail in well defined ways.

Share Improve this answer Follow

answered Nov 19, 2008 at 20:10
fizzer
13.8k ● 9 ● 43 ● 61

4

NULL checking in general is evil as it's add a small negative token to the code testability. With NULL checks everywhere you can't use "pass null" technique and it will hit you when unit testing. It's better to have unit test for the method than null check.

Check out decent presentation on that issue and unit testing in general by Misko Hevery at http://www.youtube.com/watch?v=wEhu57pih5w&feature=channel

Share Improve this answer Follow

answered Nov 19, 2008 at 18:31
Nazgob
8,573 ● 4 ● 42 ● 42

3

Older versions of Microsoft C++ (and probably others) did not throw an exception for failed allocations via new, but returned NULL. Code that had to run in both standard-conforming and older versions would have the redundant checking that you point out in your first example.

It would be cleaner to make all failed allocations follow the same code path:

```
if(obj==NULL)
    throw std::bad_alloc();
```

Share
Improve this answer
Follow

edited Nov 19, 2008 at 20:29

answered Nov 19, 2008 at 20:23
Mark Ransom
308k ● 44 ● 416 ● 647

2

It's widely known that there are procedure-oriented people (focus on doing things the right way) and results-oriented people (get the right answer). Most of us lie somewhere in the middle. Looks like you've found an outlier for procedure-oriented. These people would say "anything's possible unless you understand things perfectly; so prepare for anything." For them, what you see is done properly. For them if you change it, they'll worry because the ducks aren't all lined up.

When working on someone else's code, I try to make sure I know two things.
1. What the programmer intended
2. Why they wrote the code the way they did

For following up on Type A programmers, maybe this helps.

So "How much is enough" ends up being a social question as much as a technical question - there's no agreed-upon way to measure it.

(It drives me nuts too.)

Share
Improve this answer
Follow

edited Nov 19, 2008 at 18:05

answered Nov 19, 2008 at 17:50
dkretz
37.6k ● 13 ● 83 ● 140

Doing anything ad nauseam will usually make #1 very difficult. In one example, there was ~20 lines of actual logic and over 100 lines of null-check and handling (i.e. log it). – James Schek
Nov 19, 2008 at 18:30

2

Personally I think null testing is unnnecessary in the great majority of cases. If new fails or malloc fails you have bigger issues and the chance of recovering is just about nil in cases where you're not writing a memory checker! Also null testing hides bugs a lot in the development phases since the "null" clauses are frequently just empty and do nothing.

Share Improve this answer Follow

answered Nov 19, 2008 at 18:37
RichieHH
2,123 ● 4 ● 29 ● 31

2

When you can specify which compiler is being used, for system functions such as "new" checking for null is a bug in the code. It means that you will be duplicating the error handling code. Duplicate code is often a source of bugs because often one gets changed and the other doesn't. If you can not specify the compiler or compiler versions, you should be more defensive.

As for internal functions, you should specify the contract and make sure that contract is enforce via unit tests. We had a problem in our code a while back where we either threw an exception or returned null in case of a missing object from our database. This just made things confusing for the caller of the api so we went through and made it consistant throughout the entire code base and removed the duplicate checks.

The important thing (IMHO) is to not have duplicate error logic where one branch will never be invoked. If you can never invoke code, then you can't test it, and you will never know if it is broken or not.

Share

Improve this answer

Follow

edited Nov 19, 2008 at 19:10

answered Nov 19, 2008 at 17:55

Aaron
19.7k ● 4 ● 30 ● 23

Old versions of C++, and IIRC Embedded C++, return null from new in a similar way to malloc. – Tom Hawtin - tackline Nov 19, 2008 at 18:02

Sadly, for many years several mainstream compilers deviated from the C++ spec in this regard. AFAIK, C++ spec has *always* required the exception. Nearly all major C++ compilers are now in compliance with the spec. – James Schek Nov 19, 2008 at 18:35

I added a note about "When you can specify which compiler you use". If you are writing an application and can specify the build environment there is no reason to workaround bugs in older versions of the compiler. – Aaron Nov 19, 2008 at 19:11

2

I'd say it depends a little on your language, but I use Resharper with C# and it basically goes out of it's way to tell me "this reference could be null" in which case I add a check, if it tells me "this will always be true" for "if (null != oMyThing && ....)" then I listen to it an don't test for null.

Share Improve this answer Follow

answered Nov 19, 2008 at 21:30

dlamblin
45.3k ● 22 ● 104 ● 144

FindBugz in java provides much the same support in terms of warning "missing" and "unneccessary" null checks. – Bill Michell Nov 20, 2008 at 11:30

2

Whether to check for null or not greatly depends on the circumstances.

For example in our shop we check parameters to methods we create for null inside the method. The simple reason is that as the original programmer I have a good idea of exactly what the method should do. I understand the context even if the documentation and requirements are incomplete or less than satisfactory. A later programmer tasked with maintenance may not understand the context and may assume, wrongly, that passing null is harmless. If I know null would be harmful and I can anticipate that someone may pass null, I should take the simple step of making sure that the method reacts in a graceful way.

```
public MyObject MyMethod(object foo)
{
  if (foo == null)
  {
    throw new ArgumentNullException("foo");
  }
```

```
    // do whatever if foo was non-null
}
```

Share Improve this answer Follow

answered Nov 20, 2008 at 15:13

Mike Chess
2,798 ● 5 ● 29 ● 37

2

I only check for NULL when I know what to do when I see NULL. "Know what to do" here means "know how to avoid a crash" or "know what to tell the user besides the location of the crash". For example, if malloc() returns NULL, I usually have no option but to abort the program. On the other hand, if fopen() returns NULL, I can let the user know the file name that could not be open and may be errno. And if find() returns end(), I usually know how to continue without crashing.

Share Improve this answer Follow

answered Nov 21, 2008 at 19:54

user3458

1

Lower level code should check use from higher level code. Usually this means checking arguments, but it can mean checking return values from upcalls. Upcall arguments need not be checked.

The aim is to catch bugs in immediate and obvious ways, as well as documenting the contract in code that does not lie.

Share Improve this answer Follow

answered Nov 19, 2008 at 18:05

Tom Hawtin - tackline
147k ● 30 ● 221 ● 312

1

I don't think it's *bad* code. A fair amount of Windows/Linux API calls return NULL on failure of some sort. So, of course, I check for failure in the manner the API specifies. Usually I wind up passing control flow to an error module of some fashion instead of duplicating error-handling code.

Share Improve this answer Follow

answered Nov 19, 2008 at 21:24

Paul Nathan
40.2k ● 30 ● 120 ● 215

If I receive a pointer that is not guaranteed by language to be not null, and am going to de-reference it in a way that null will break me, or pass out put my function where I said I wouldn't produce NULLs, I check for NULL.

1

It is not just about NULLs, a function should check pre- and post-conditions if possible.

It doesn't matter at all if a contract of the function that gave me the pointer says it'll never produce nulls. We all make bugs. There's a good rule that a program shall fail early and often, so instead of passing the bug to another module and have it fail, I'll fail in place. Makes things so much easier to debug when testing. Also in critical systems makes it easier to keep the system sane.

Also, if an exception escapes main, stack may not be rolled up, preventing destructors from running at all (see C++ standard on terminate()). Which may be serious. So leaving bad_alloc unchecked can be more dangerous than it seems.

Fail with assert vs. fail with a run time error is quite a different topic.

Checking for NULL after new() if standard new() behavior has not been altered to return NULL instead of throwing seems obsolete.

There's another problem, which is that even if malloc returned a valid pointer, it doesn't yet mean you have allocated memory and can use it. But that is another story.

Share
Improve this answer
Follow

edited Nov 20, 2008 at 14:59

answered Nov 20, 2008 at 14:40
n-alexander
15.3k ● 13 ● 44 ● 43

"even if malloc returned a valid pointer, it doesn't yet mean you have allocated memory and can use it" - technically, I think the C and C++ standards say it does mean that, which is why linux has to core dump ("cop a guilty plea") at the point it fails to commit the memory.
– Steve Jessop Nov 21, 2008 at 12:51

I think it is different. Malloc() fails to guarantee memory, unlike new(). Then when you run into trouble you get killed, and this si quite besides the language. Your process might as well have been written in Fortran - Linux kernel doesn't know or care. So this is not standard enforcing.
– n-alexander Nov 21, 2008 at 17:05

1

My first problem with this, is that it leads to code which is littered with null checks and the likes. It hurts readability, and i'd even go as far as to say that it hurts maintainability because it really is easy to forget a null check if you're writing a piece of code where a certain reference really should never be null. And you just know that the null checks will be missing in some places. Which actually makes debugging harder than it needs to be. Had the original exception not been caught and replaced with a faulty return value, then we would've gotten a valuable exception object with an informative stacktrace. What does a missing null check give you? A

NullReferenceException in a piece of code that makes you go: wtf? this reference should never be null!

So then you need to start figuring out how the code was called, and why the reference could possibly be null. This can take a lot of time and really hurts the efficiency of your debugging efforts. Eventually you'll figure out the real problem, but odds are that it was hidden pretty deeply and you spent a lot more time searching for it than you should have.

Another problem with null checks all over the place is that some developers don't really take the time to properly think about the real problem when they get a NullReferenceException. I've actually seen quite a few developers just add a null check above the code where the NullReferenceException occurred. Great, the exception no longer occurs! Hurray! We can go home now! Umm… how bout 'no you can't and you deserve an elbow to the face'? The real bug might not cause an exception anymore, but now you probably have missing or faulty behavior… and no exception! Which is even more painful and takes even more time to debug.

Share Improve this answer Follow

answered Sep 20, 2013 at 7:04
Cornel Marian
2,503 ● 23 ● 28

0

At first, this seemed like a strange question: `null` checks are great and a valuable tool. Checking that `new` returns `null` is definitely silly. I'm just going to ignore the fact that there are languages that allow that. I'm sure there are valid reasons, but I really don't think I can handle living in that reality :) All kidding aside, it seems like you should at least have to specify that the `new` should return `null` when there isn't enough memory.

Anyway, checking for `null` where appropriate leads to cleaner code. I'd go so far as to say that never assigning function parameters default values is the next logical step. To go even further, returning empty arrays, etc. where appropriate leads to even cleaner code. It is nice to not have to worry about getting `null`s except where they are logically meaningful. Nulls as error values are better avoided.

Using asserts is a really great idea. Especially if it gives you the option of turning them off at runtime. Plus, it is a more explicitly contractual style :)

Share Improve this answer Follow

answered Mar 25, 2011 at 1:48
wprl
25.4k ● 11 ● 57 ● 70