

# Design by contract using assertions or exceptions? [closed]

Asked 16 years, 3 months ago   Modified 7 years, 11 months ago

Viewed 28k times



134



**Closed.** This question is [opinion-based](#). It is not currently accepting answers.



**Want to improve this question?** Update the question so it can be answered with facts and citations by [editing this post](#).

Closed 6 years ago.

[Improve this question](#)

When programming by contract a function or method first checks whether its preconditions are fulfilled, before starting to work on its responsibilities, right? The two most prominent ways to do these checks are by `assert` and by `exception`.

1. `assert` fails only in debug mode. To make sure it is crucial to (unit) test all separate contract preconditions to see whether they actually fail.
2. `exception` fails in debug and release mode. This has the benefit that tested debug behavior is identical to

release behavior, but it incurs a runtime performance penalty.

Which one do you think is preferable?

See related question [here](#)

exception

assert

design-by-contract

Share

Improve this question

Follow

edited May 23, 2017 at 12:34



Community Bot

1 • 1

asked Sep 22, 2008 at 19:57



andreas buykx

12.9k • 11 • 64 • 76

- 
- 3 The entire point behind design by contract is that you don't need to (and arguably shouldn't) verify preconditions at runtime. You verify the input prior to passing it into the method with the preconditions, that's how you respect *your* end of the contract. If input is invalid or violates your end of the contract, the program will usually fail anyway through its normal course of actions (which you want). – [void.pointer](#) Apr 9, 2012 at 16:48
- 

Nice question, but I think you should really switch the accepted answer (as the votes show, too)! – [DaveFar](#) Apr 25, 2012 at 9:30

---

Forever later, I know, but should this question actually have the c++ tag? I was looking for this answer, to use in another language (Delpih) and I can't imagine any language that feature exceptions and assertions that wouldn't follow the

same rules. (Still learning Stack Overflow guidelines.)

– [Eric G](#) Sep 13, 2012 at 1:00

Very succinct response given [in this response](#): "In other words, exceptions address the robustness of your application while assertions address its correctness." – [Shmuel Levine](#)

Jun 11, 2015 at 16:04 

## 14 Answers

Sorted by:

Highest score (default)



202



The rule of thumb is that you should use assertions when you are trying to catch your own errors, and exceptions when trying to catch other people's errors. In other words, you should use exceptions to check the preconditions for the public API functions, and whenever you get any data that are external to your system. You should use asserts for the functions or data that are internal to your system.

Share Improve this answer

answered Sep 22, 2008 at 20:06

Follow



[Dima](#)

39.3k ● 14 ● 78 ● 116

1 what about serialize/deserialize sitting in different modules/applications and eventually going out of sync? I mean on reader part it is always my mistake if I am trying to read the things the wrong way so I tend to use asserts, but on the other hand I have external data, which can eventually change format without notice. – [Slava](#) Oct 24, 2014 at 10:17


1 If the data is external, then you should use exceptions. In this particular case you should probably also catch those exceptions, and handle them in some reasonable way, rather than just letting your program die. Also, my answer is a rule

of thumb, not a law of nature. :) So you have to consider each case individually. – [Dima](#) Oct 24, 2014 at 15:36

---

If your function  $f(\text{int}^* x)$  contains a line  $x \rightarrow \text{len}$  in it, then  $f(v)$  where  $v$  is proven to be null is guaranteed to crash. Furthermore, if even earlier on  $v$  is proven to be null yet  $f(v)$  is proven to be called, you have a logical contradiction. It's the same as having  $a/b$  where  $b$  is ultimately proven to be 0. Ideally, such code should fail to compile. Turning off assumption checks is completely foolish unless the issue is the cost of the checks, because it obscures the location where an assumption was violated. It must at least be logged. You should have a restart-on-crash design anyway. – [Rob](#) Jan 7, 2015 at 22:28

---

Since we should use exceptions for *external* errors and assertions for *internal* errors, we should either use exceptions or assertions for *preconditions*, and always use assertions for *postconditions* and *invariants*, right? – [Géry Ogam](#) Apr 9, 2022 at 14:32 

---



40



Disabling assert in release builds is like saying "I will never have any issues whatsoever in a release build", which is often not the case. So assert shouldn't be disabled in a release build. But you don't want the release build crashing whenever errors occur either, do you?



So use exceptions and use them well. Use a good, solid exception hierarchy and ensure that you catch and you can put a hook on exception throwing in your debugger to catch it, and in release mode you can compensate for the error rather than a straight-up crash. It's the safer way to go.

Share Improve this answer

edited Oct 30, 2008 at 0:43

Follow



Adam Bellaire

110k ● 19 ● 152 ● 165

answered Sep 22, 2008 at 20:03



coppo

14.5k ● 5 ● 61 ● 73

- 
- 4 Assertions are useful at the very least in cases where checking correctness would be either inefficient or inefficient to implement properly. – [Casebash](#) Oct 25, 2009 at 6:24
- 
- 96 The point in assertions is not to correct errors, but to alert the programmer. Keeping them enabled in release builds is useless for *that* reason: What would you have gained by having an assert firing? The developer won't be able to jump in and debug it. Assertions are a debugging aid, they are not a replacement for exceptions (and nor are exceptions are replacement for assertions). Exceptions alert the program to an error condition. Assert alerts the developer.  
– [Stack Overflow is garbage](#) Oct 25, 2009 at 13:50
- 
- 12 But an assertion should be used when the internal data has been corrupted past fixing - if an assertion triggers, you can make no assumptions about the state of the program because it means something is /wrong/. If an assertion has gone off, you can't assume any data is valid. That's why a release build should assert - not to tell the programmer where the problem is, but so that the program can shut down and not risk bigger problems. The program should just do what it can to facilitate recovery later, when the data can be trusted. – [coppo](#) Oct 26, 2009 at 0:09
- 
- 5 @jalf, Although you cannot put a hook into your debugger in release builds, you can leverage logging so that developers see the information relevant to your assertion failing. In this document ([martinfowler.com/ieeeSoftware/failFast.pdf](http://martinfowler.com/ieeeSoftware/failFast.pdf)), Jim Shore points out, "Remember, an error that occurs at the

customer's site made it through your testing process. You'll probably have trouble reproducing it. These errors are the hardest to find, and a well-placed assertion explaining the problem could save you days of effort." – [StriplingWarrior](#)  
Jun 7, 2010 at 21:28

---

- 5 Personally I prefer asserts for design by contract approaches. Exceptions are defensive and are doing the argument checking inside the function. Also, dbc preconditions don't say "I won't work if you use values out of the working range" but "I won't guarantee to provide the right answer, but I still may do". The asserts provide the developer with feedback that they are calling a function with a condition breach, but don't stop them from using it if they feel they know better. The breach could cause exceptions to occur, but I see that as a different thing. – [Matt\\_JD](#) May 22, 2011 at 7:45
- 



25

The principle I follow is this: If a situation can be realistically avoided by coding then use an assertion. Otherwise use an exception.



Assertions are for ensuring that the Contract is being adhered to. The contract must be fair, so that client must be in a position to ensure it complies. For example, you can state in a contract that a URL must be valid because the rules about what is and isn't a valid URL are known and consistent.



Exceptions are for situations that are outside the control of both the client and the server. An exception means that something has gone wrong, and there's nothing that could have been done to avoid it. For example, network

connectivity is outside the applications control so there is nothing that can be done to avoid a network error.

I'd like to add that the Assertion / Exception distinction isn't really the best way to think about it. What you really want to be thinking about is the contract and how it can be enforced. In my URL example above that best thing to do is have a class that encapsulates a URL and is either Null or a valid URL. It is the conversion of a string into a URL that enforces the contract, and an exception is thrown if it is invalid. A method with a URL parameter is much clearer than a method with a String parameter and an assertion that specifies a URL.

Share Improve this answer

edited Sep 23, 2008 at 20:24

Follow

answered Sep 22, 2008 at 20:13



Ged Byrne

801 ● 6 ● 10



7

Asserts are for catching something a developer has done wrong (not just yourself - another developer on your team also). If it's reasonable that a user mistake could create this condition, then it should be an exception.



Likewise think about the consequences. An assert typically shuts down the app. If there is any realistic expectation that the condition could be recovered from, you should probably use an exception.

On the other hand, if the problem can **only** be due to a programmer error then use an assert, because you want to know about it as soon as possible. An exception might be caught and handled, and you would never find out about it. And yes, you should disable asserts in the release code because there you want the app to recover if there is the slightest chance it might. Even if the state of your program is profoundly broken the user just might be able to save their work.

Share Improve this answer

edited Sep 25, 2008 at 21:24

Follow

answered Sep 22, 2008 at 21:39



DJClayworth

26.8k ● 9 ● 58 ● 80



5



It is not exactly true that "assert fails only in debug mode."

In *Object Oriented Software Construction, 2nd Edition* by Bertrand Meyer, the author leaves a door open for checking preconditions in release mode. In that case, what happens when an assertion fails is that... an assertion violation exception is raised! In this case, there is no recovery from the situation: something useful could be done though, and it is to automatically generate an error report and, in some cases, to restart the application.

The motivation behind this is that preconditions are typically cheaper to test than invariants and



postconditions, and that in some cases correctness and "safety" in the release build are more important than speed. i.e. For many applications speed is not an issue, but **robustness** (the ability of the program to behave in a safe way when its behaviour is not correct, i.e. when a contract is broken) is.

Should you always leave precondition checks enabled? It depends. It's up to you. There is no universal answer. If you're making software for a bank, it might be better to interrupt execution with an alarming message than to transfer \$1,000,000 instead of \$1,000. But what if you're programming a game? Maybe you need all the speed you can get, and if someone gets 1000 points instead of 10 because of a bug that the preconditions didn't catch (because they're not enabled), tough luck.

In both cases you should ideally have caught that bug during testing, and you should do a significant part of your testing with assertions enabled. What is being discussed here is what is the best policy for those rare cases in which preconditions fail in production code in a scenario which was not detected earlier due to incomplete testing.

To summarize, **you can have assertions and still get the exceptions automatically**, if you leave them enabled - at least in Eiffel. I think to do the same in C++ you need to type it yourself.

See also: [When should assertions stay in production code?](#)

Share Improve this answer

edited May 23, 2017 at 12:25

Follow



Community Bot

1 • 1

answered Dec 29, 2008 at 15:10



Daniel Daranas

22.6k • 9 • 65 • 121

- 
- 1 Your point is definitely valid. The SO didn't specify a particular language -- in the case of C# the standard assert is `System.Diagnostics.Debug.Assert`, which *does* only fail in a Debug build, and will be removed at compile time in a Release build. – [yoyo](#) Apr 14, 2016 at 23:12
- 



3



I outlined my view on the state of the matter here: [How do you validate an object's internal state?](#) . Generally, assert your claims and throw for violation by others. For disabling asserts in release builds, you can do:

- Disable asserts for expensive checks (like checking whether a range is ordered)
- Keep trivial checks enabled (like checking for a null pointer or a boolean value)

Of course, in release builds, failed assertions and uncaught exceptions should be handled another way than in debug builds (where it could just call `std::abort`). Write a log of the error somewhere (possibly into a file), tell the customer that an internal error occurred. The customer will be able to send you the log-file.

Share Improve this answer

edited May 23, 2017 at 11:46

Follow



Community Bot

1 • 1

answered Dec 29, 2008 at 15:25



Johannes Schaub - litb

506k • 131 • 917 • 1.2k



2



There was a huge [thread](#) regarding the enabling/disabling of assertions in release builds on `comp.lang.c++.moderated`, which if you have a few weeks you can see how varied the opinions on this are. :)

Contrary to [coppro](#), I believe that if you are not sure that an assertion can be disabled in a release build, then it should not have been an assert. Assertions are to protect against program invariants being broken. In such a case, as far as the client of your code is concerned there will be one of two possible outcomes:

1. Die with some kind of OS type failure, resulting in a call to abort. (Without assert)
2. Die via a direct call to abort. (With assert)

There is no difference to the user, however, it's possible that the assertions add an unnecessary performance cost in the code that is present in the vast majority of runs where the code doesn't fail.

The answer to the question actually depends much more on who the clients of the API will be. If you are writing a library providing an API, then you need some form of

mechanism to notify your customers that they have used the API incorrectly. Unless you supply two versions of the library (one with asserts, one without) then assert is very unlikely the appropriate choice.

Personally, however, I'm not sure that I would go with exceptions for this case either. Exceptions are better suited to where a suitable form of recovery can take place. For example, it may be that you're trying to allocate memory. When you catch a 'std::bad\_alloc' exception it might be possible to free up memory and try again.

Share Improve this answer

Follow

edited May 23, 2017 at 12:17



Community Bot

1 • 1

answered Sep 23, 2008 at 8:17



Richard Corden

21.7k • 9 • 61 • 87



1



you're asking about the difference between design-time and run-time errors.

asserts are 'hey programmer, this is broken' notifications, they're there to remind you of bugs you wouldn't have noticed when they happened.



exceptions are 'hey user, somethings gone wrong' notifications (obviously you can code to catch them so the user never gets told) but these are designed to occur at run time when Joe user is using the app.

So, if you think you can get all your bugs out, use exceptions only. If you think you can't..... use exceptions. You can still use debug asserts to make the number of exceptions less of course.

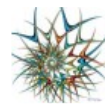
Don't forget that many of the preconditions will be user-supplied data, so you will need a good way of informing the user his data was no good. To do that, you'll often need to return error data down the call stack to the bits he is interacting with. Asserts will not be useful then - doubly so if your app is n-tier.

Lastly, I'd use neither - error codes are far superior for errors you think will occur regularly. :)

Share Improve this answer

answered Sep 22, 2008 at 20:05

Follow



[gbjbaanb](#)

52.6k ● 12 ● 110 ● 154



0



I prefer the second one. While your tests may have run fine, [Murphy](#) says that something unexpected will go wrong. So, instead of getting an exception at the actual erroneous method call, you end up tracing out a NullPointerException (or equivalent) 10 stack frames deeper.



Share Improve this answer

answered Sep 22, 2008 at 20:08

Follow



[jdmichal](#)

11.1k ● 4 ● 46 ● 42



0

The previous answers are correct: use exceptions for public API functions. The only time you might wish to bend this rule is when the check is computationally expensive. In that case, you **can** put it in an assert.



If you think violation of that precondition is likely, keep it as an exception, or refactor the precondition away.



Share Improve this answer

answered Sep 22, 2008 at 20:51

Follow



Mike Elkins

1,438 ● 1 ● 11 ● 13



0

You should use both. Asserts are for your convenience as a developer. Exceptions catch things you missed or didn't expect during runtime.



I've grown fond of [glib's error reporting functions](#) instead of plain old asserts. They behave like assert statements but instead of halting the program, they just return a value and let the program continue. It works surprisingly well, and as a bonus you get to see what happens to the rest of your program when a function doesn't return "what it's supposed to". If it crashes, you know that your error checking is lax somewhere else down the road.



In my last project, I used these style of functions to implement precondition checking, and if one of them failed, I would print a stack trace to the log file but keep on running. Saved me tons of debugging time when other

people would encounter a problem when running my debug build.

```
#ifdef DEBUG
#define RETURN_IF_FAIL(expr)      do {
\
\   if (!(expr))
\
\   {
\
\       fprintf(stderr,
\
\           "file %s: line %d (%s): precondition `%s'
failed.", \
\           __FILE__,
\
\           __LINE__,
\
\           __PRETTY_FUNCTION__,
\
\           #expr);
\
\       ::print_stack_trace(2);
\
\       return;
\
\   };
\           } while(0)
#define RETURN_VAL_IF_FAIL(expr, val) do {
\
\   if (!(expr))
\
\   {
\
\       fprintf(stderr,
\
\           "file %s: line %d (%s): precondition `%s'
failed.", \
\           __FILE__,
\
\           __LINE__,
\
\
```

If I needed runtime checking of arguments, I'd do this:

```
char *doSomething(char *ptr)
{
    RETURN_VAL_IF_FAIL(ptr != NULL, NULL); //
    same as assert(ptr != NULL), but returns NULL if
    it fails.                                //

    Goes away when debug off.

    if( ptr != NULL )
    {
        ...
    }

    return ptr;
}
```

Share Improve this answer

answered Sep 22, 2008 at 21:23

Follow



[indiv](#)

17.8k ● 6 ● 64 ● 83

---

I don't think I've seen in OP question anything related to C++. I believe it should not be included in your answer.

– [ForceMagic](#) Aug 7, 2013 at 17:26

---

@ForceMagic: The question had the C++ tag in 2008 when I posted this answer, and in fact the C++ tag was removed only 5 hours ago. Regardless, the code illustrates a language-independent concept. – [indiv](#) Aug 7, 2013 at 17:45

---



I tried synthesising several of the other answers here with my own views.

0





Use assertions for cases where you want to disable it in production, erring toward leaving them in. The only real reason to disable in production, but not in development, is to speed up the program. In most cases, this speed up won't be significant, but sometimes code is time critical or the test is computationally expensive. If code is mission critical, then exceptions may be best despite the slow down.

If there is any real chance of recovery, use an exception as assertions aren't designed to be recovered from. For example, code is rarely designed to recover from programming errors, but it is designed to recover from factors such as network failures or locked files. Errors should not be handled as exceptions simply for being outside the control of the programmer. Rather, the predictability of these errors, compared to coding mistakes, makes them more amiable to recovery.

Re argument that it is easier to debug assertions: The stack trace from a properly named exception is as easy to read as an assertion. Good code should only catch specific types of exceptions, so exceptions should not go unnoticed due to being caught. However, I think Java sometimes forces you to catch all exceptions.

Share Improve this answer

edited Oct 25, 2009 at 7:40

Follow

answered Oct 25, 2009 at 7:12



Casebash

118k ● 92 ● 251 ● 353



0



The rule of thumb, to me, is that use assert expressions to find internal errors and exceptions for external errors. You can benefit much from the following discussion by Greg from [here](#).

Assert expressions are used to find programming errors: either errors in the program's logic itself or in errors in its corresponding implementation. An assert condition verifies that the program remains in a defined state. A "defined state" is basically one that agrees with the program's assumptions. Note that a "defined state" for a program need not be an "ideal state" or even "a usual state", or even a "useful state" but more on that important point later.

To understand how assertions fit into a program, consider a routine in a C++ program that is about to dereference a pointer. Now should the routine test whether the pointer is NULL before the dereferencing, or should it assert that the pointer is not NULL and then go ahead and dereference it regardless?

I imagine that most developers would want to do both, add the assert, but also check the pointer for a NULL value, in order not to crash should the

asserted condition fail. On the surface, performing both the test and the check may seem the wisest decision

Unlike its asserted conditions, a program's error handling (exceptions) refers not to errors in the program, but to inputs the program obtains from its environment. These are often "errors" on someone's part, such as a user attempting to login to an account without typing in a password. And even though the error may prevent a successful completion of program's task, there is no program failure. The program fails to login the user without a password due to an external error - an error on the user's part. If the circumstances were different, and the user typed in the correct password and the program failed to recognize it; then although the outcome would still be the same, the failure would now belong to the program.

The purpose of error handling (exceptions) is two fold. The first is to communicate to the user (or some other client) that an error in program's input has been detected and what it means. The second aim is to restore the application after the error is detected, to a well-defined state. Note that the program itself is not in error in this situation. Granted, the program may be in a non-ideal state, or even a state in which can do nothing useful, but there is no programming

errorl. On the contrary, since the error recovery state is one anticipated by the program's design, it iss one that the program can handle.

PS: you may want to check out the similar question: [Exception Vs Assertion](#).

Share Improve this answer

Follow

edited May 23, 2017 at 11:54



Community Bot

1 • 1

answered Jan 5, 2014 at 11:25



herohuyongtao

50.6k • 30 • 137 • 176

See also [this question](#):



-1



I some cases, asserts are disabled when building for release. You may not have control over this (otherwise, you could build with asserts on), so it might be a good idea to do it like this.

The problem with "correcting" the input values is that the caller will not get what they expect, and this can lead to problems or even crashes in wholly different parts of the program, making debugging a nightmare.

I usually throw an exception in the if-statement to take over the role of the assert in case they are disabled

```
assert(value>0);  
if(value<=0) throw new  
ArgumentOutOfRangeException("value");  
//do stuff
```

Share Improve this answer

Follow

edited May 23, 2017 at 12:02



Community Bot

1 ● 1

answered Sep 22, 2008 at 20:18



Rik

29.2k ● 14 ● 51 ● 69

---