P2861R0
2023-05-18

John Lakos: jlakos@bloomberg.net

# The Lakos Rule
## *Narrow Contracts* and `noexcept` Are Inherently Incompatible


## Abstract
A *contract* in the C++ Standard Library, is a specification, for a given function, of what *subset* of syntactically valid input and accessible state is required to invoke that function so as to have *defined* (i.e., not *undefined) behavior* and, in particular, the *essential behavior* that function promises to deliver when invoked *in contract*. Functions having at least one syntactically valid input-and-state combination for which the behavior is undefined are said to have *narrow contracts*. As of C++11, a new keyword, `noexcept`, was added to the language. When used as a function specifier, this keyword effectively codifies *essential behavior* for the entire syntactically valid domain of the function to be, in addition to any other explicitly specified requirements, "throws nothing" — a contradiction. Since C++11, the *Lakos Rule*, as reflected in the Standard Library, effectively prohibits placing the `noexcept` specifier on any function that would otherwise have a *narrow contract*. This paper explains why that rule was, is, and likely always will be a solid best practice, especially in the C++ Standard Library.


## Table of Contents

# 0 REVISIONS

R0: Initial paper submission

# 1 INTRODUCTION

The use of *narrow contracts* — those having one or more *preconditions* — has been an integral part of practical, efficient software engineering since long before the first “C with Classes” program was even on the drawing board. In the 1980s, legends like Barbara Liskov and Bertrand Meyer were focused on how to structure hierarchies of telescoping preconditions.

In particular, the first edition of the C programming language, published February 22, 1978, shows the classic program to print "hello, world":

```
main() {

    printf("hello, world\n");  // well-defined behavior

}
```

The function `printf` is not valid with every syntactically valid input. Even ignoring the lax type safety in classic C, adding a simple percent sign (`%`) in the quoted string would result in *undefined behavior*:

```
main() {
    printf("hello, worl%d\n");  // undefined behavior
}
```

Hence, `printf` had a *narrow contract*. Even today in C++, `printf` has a narrow contract but not *as* narrow as it was in the early days. Over the years, behavior that was previously undefined, such as `%x`, `%Lx`, and `%LLx` has taken on meaning where previously there was none. This sort of backward-compatible extensibility *over time* is what makes truly *narrow* contracts inherently essential to the effective maintainability and enhancement of virtually every successful, widely used library we write.

In what follows, we'll start by precisely defining important terms that we use routinely when discussing contracts, which will include two kinds of *undefined behavior* (UB) — *library* UB and *language* UB (the distinction between which is

not *yet* properly reflected within the Standard). At this point, we'll restate the formal proof that functions having a nonthrowing exception specification — *by definition* — cannot have a narrow contract, culminating in the Lakos Rule.

We'll then explore the value of extending a particular narrow contract — one that initially *throws nothing* — over several versions of a library but this time from a C++ perspective. After that, we'll examine the tangled web we would have woven had we added `noexcept` to that first nonthrowing function on its first version.

Next, we'll take a look how narrow contracts interact with contract checking and justify the general need, at least for some, for throwing contract-violation handlers. In particular, we'll explore throwing on a contract-violation as the primary means of (1) temporary continuation, if not full recovery, and (2) maximally effective, efficient, and portable *negative testing*.

Then, we will turn our attention the adverse interaction that liberal application of `noexcept` specifiers has on software designs that make use of exceptions to communicate uncommon or unexpected (e.g., logic) errors. After that, we'll take a look at various practical ways the `noexcept` specifier can be used (and misused) to achieve superior *algorithmic* performance when called from a *generic context* that demands more than just the basic exception-safety guarantee, along with some other niche use cases.

Importantly, although widespread systemic use of the `noexcept` specifier can somewhat reliably reduce generated code size, *no* legitimate theory or empirical evidence suggests that *runtime* performance is ever measurably, let alone significantly, improved; copious experimental data and basic common sense, however, suggest otherwise.

At this point, we'll change gears and talk about the Standard Library's role as the foundation for virtually all C++ development worldwide. As such, the Standard cannot reasonably support any one particular set of design principles (i.e., one developer's universe) and must instead and to the extent practicable support all of them without undue judgment.

After that, we'll revisit the Lakos Rule and discuss its implications. We'll then postulate what an exception to the Lakos Rule might look like and then test our hypothesis on what were initially deemed possible exceptions to the Lakos Rule to see if they meet a set of four criteria by which all potential exceptions are to be validated. At the conclusion of this section, we'll then exhibit the only known exception to the Lakos Rule, i.e., the exception that proves the rule.

Finally, in that light, we provide recommendations and justification for effective practical use of the `noexcept` specifier for (1) the C++ Standard Library

(specification), (2) concrete implementations of the Standard Library, (3) third-party libraries, and (4) end-user libraries. Spoiler alert: Absent a vanishingly rare and compelling engineering reason to do otherwise, deviating from the Lakos Rule is invariably and absolutely a terribly bad idea, especially within the C++ Standard Library specification.

## 2 SYNOPSIS

This paper is tutorial in nature, and thus is long with many supporting examples. To facilitate a much quicker read, we provide here a quick synopsis of the paper.

In Section 3, we start by defining some key terms.

- Precondition — a requirement (expressed or implied) that limits the circumstances under which a contract can be considered binding.
- Essential behavior — that which must result from a properly implemented function provided all preconditions for that function are met.
- Contract — a bilateral agreement (typically written in a natural language) between a function's implementation and any caller of that function. It must incorporate at least (1) any nonobvious preconditions necessary for successfully invocation and (2) the essential behavior that the function promises to deliver, provided that all preconditions (even implicit ones) are met.
- Narrow contract — a contract with preconditions.
- Wide contract — a contract without preconditions.
- Conforming implementation — one that accurately implements the contract.
- Undefined behavior (UB) — behavior for which there are no requirements.
- Library UB — behavior resulting from invoking a nonstandard library function in violation of one (or more) preconditions.
- Language UB — behavior resulting from a failure to satisfy the constraints of the language or the Standard Library.

We point out, in Section 4, that by adding a `noexcept` specifier to a function with a *narrow contract*, we are effectively widening that *contract* by imposing essential behavior on all possible syntactically valid input combinations. We discuss the importance of functions with *narrow contracts* and revise both the *Liskov Substitutability Principle* and Bertrand Meyer's *Design-by-Contract*. We then show that, where a function is declared `noexcept`, we cannot override that function with one that can throw. Similarly, we cannot subsequently modify a function's *contract* with one that is permitted to throw without breaking backward compatibility. The final point of Section 4 is that many libraries perform defensive input checks when built in debug mode, throwing

an exception if called *out of contract*, and skip those checks when built in release mode. Marking those functions `noexcept` would prevent library implementations from providing such checks.

In section 5, we consider interactions with the upcoming Contracts MVP facility in C++. A number of other languages currently have contract checking and, in the event of contract violation, use an exception to indicate the fact. This enables an application to recover from the problem (e.g., retrying an operation that timed out), to gracefully degrade (e,g., in the rendering of a video game), or to shut down in a controlled manner (e.g., closing any open orders in a stock market trading system). When the checked function is declared `noexcept`, the remaining options are to ignore the fact or to terminate, both of which can be a serious problem in many environments. Having a `noexcept` specifier on a *narrow-contract* function with a Contracts MVP check would also remove the only viable means of thoroughly testing the function's contracts check, namely to have a handler that throws an exception.

Section 6 reprises some of the potential pitfalls of `noexcept`, described in *Embracing Modern C++ Safely*,[1] namely overly strong *contract* guarantees and accidental termination.

Section 7 then looks at some of the common uses and misuses of the `noexcept` specifier.

- Use: declaring move operations nonthrowing, which enables algorithms offering exception safety guarantees to perform additional optimizations when it is known that an exception cannot be emitted. This legitimate use is the reason `noexcept` was introduced into the language in C++11.
- Use: wrapping a type to force the Standard Library to apply its `noexcept` optimizations even though, in reality, that type can throw on move.
- Borderline misuse: to avoid writing exception handler functions in callback frameworks.
- Misuse: as a form of documentation or as a substitute for documenting possible exceptions in the *contract*.
- Misuse: reducing object code size on the (typically false) assumption of faster performance.
- Misuse: attempting to increase performance (despite that `noexcept` will, on its own, make no difference whatsoever).

Section 8 points out that the C++ Standard must cater to the *multiverse*, i.e., all the industries that use C++, each with their own needs and priorities. The Standard should, without any value judgement, ratify the widest set of use cases without overly inconveniencing the typical case. The limitations that

[1] [Lakos22a]

would be imposed by the removal of the Lakos Rule are in conflict with this principle.

Section 9 defines the Lakos Rule as "Narrow contracts and `noexcept` are inherently incompatible." It also discusses possible exceptions to the rule, a set of key tests that should be applied when considering any exception, and how `swap` is the only valid exception (noting that move operations are a legitimate use for `noexcept`, which does not break the Lakos Rule).

Section 10 states that application of the Lakos Rule is beneficial for third-party and end-user libraries in addition to the Standard Library and its implementations.

# 3 TERMINOLOGY

A *contract* governing the behavior of a function, member function, operator, constructor, or lambda — henceforth referred to collectively as *function* — is a bilateral agreement (typically written in a natural language) between that function's implementation and any caller of that function. A function's contract must, at a minimum, incorporate (1) any nonobvious *preconditions* necessary for successfully invoking that function and (2) the *essential behavior* that the function promises to deliver, provided that *all* preconditions (even implicit ones) are met.

## 3.1 Essential and Implementation-Defined Behaviors

We define *essential behavior* to be that which must result from a properly implemented function provided that all *preconditions* for that function are met.

Consider a function `average` that takes two integers, `a` and `b`, and returns their mean rounded up to the nearest integer:

```
int average(int x, int y);
    // Return the closest integer, `z`, to the mean of `x` and `y`; if `z` is
    // not unique, return the larger one.
```

The contract for the `average` function above has no *preconditions*. A *conforming implementation*, one that accurately implements the contract, must therefore work correctly — i.e., satisfy its contractually promised *essential behavior* for *every* combination of syntactically valid inputs.

May a *conforming implementation* do something *other* than what is stated as its essential behavior? No! If this function, for any reason, fails to return the rounded mean of `a` and `b`, the implementation of the function is *nonconforming*. Perhaps the implementer didn't understand the contract or made an inadvertent coding error. To become conforming, either the implementation or the contract would have to change.

May a *conforming implementation* do something in addition to the *essential* behavior stated in the contract. Yes! In addition to *essential behavior*, behavior that is *unspecified* can also occur, which, for a function that is called *in contract* (i.e., all of its preconditions are satisfied), is referred to by the Standard as *implementation-defined.*[2]

Suppose the contract for the `average` function above did not specify what happens when we have two equally close results:

```
int average2(int x, int y);
    // Return a closest integer `z`, to the mean of `x` and `y.
```

The contract for `average2` implicitly reads:

```
int average2(int x, int y);
    // Return a closest integer,`z`, to the mean of `x` and `y;
    // of the two values, which value is returned when `z` is not unique is
    // implementation defined.
```

When calling `average2(3, 6)`, a conforming implementation may return either `4` or `5`, whereas a call to the original `average(3, 6)` is duty bound to return `5` and nothing else.

The example above might seem contrived, but nonessential behavior is inherent in many, if not most, higher-level function contracts. For example, consider a pair of functions, `sortOnX` and `stableSortOnX`, that each take a range of x-y integer `Point` objects and sort them in ascending order according to just their `x` values:

```
struct Point { int x, y; };  // data element to be sorted

void sortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`.

void stableSortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`, preserving the
    // original relative order of objects of equal `x` value.
```

Notice that the essential behavior of `sortOnX` leaves room for variation in the result, whereas no such variation is allowed for `stableSortOnX`. Still, `sortOnX` must return its result is some order that satisfies the stated *essential behavior*.

---

[2] The difference in the Standard's definition of *unspecified behavior* and *implementation-defined behavior* is that *unspecified* behavior is permitted to include *undefined behavior,* whereas *implementation-defined behavior* is intended to be selected from an enumerable set of alternatives that explicitly does not include *undefined behavior*. We will talk more about *undefined behavior* shortly.

For concreteness, consider a Point sequence, `a`, consisting of three points:

```
static Point a[] = { { 9, 1 }, { 9, 2 }, { 8, 3 } };
```

Calling `sortOnX(a, a+3)` *must* produce either one of two results:

```
(a) S[]: (8, 3), ( 9, 2 ), ( 9, 1 )
(b) S[]: (8, 3), ( 9, 1 ), ( 9, 2 )
```

Calling `stableSortOnX(a, a + 3)`, however, *must* produce only the latter.

May a *conforming implementation* additionally do something that is entirely unrelated to essential behavior? Yes. While *essential behavior* governs what *must* happen to fulfill the contract, it simply cannot possibly govern everything that must *not* happen. Imagine you hire a carpenter to build shelves for your new office. The carpenter's contract is as follows: “If you give me X dollars, I'll build Y shelves for you.” You agree and give the carpenter X dollars. In a few hours, the carpenter returns with freshly built shelves. Assuming the shelves are satisfactory, the essential behavior was met. The carpenter's behavior was conforming.

But what if the carpenter also mailed you a thank-you card? Is the carpenter's behavior still conforming? None of the *essential behavior* for `average`, `average2`, `sortOnX`, or `stableSortOnX` above stated anything about output (or the lack thereof). Is it possible that a *conforming implementation* for `average` might occasionally print “Thank You!” to standard output? That behavior might not be what you expected or even wanted, but even such an unusual implementation of `average` — by definition — must be considered *conforming*.

Again, a *conforming implementation* requires that, if on invocation of a function, all of its *preconditions* (expressed or implied) are satisfied, all of the stated *essential behavior* will occur. Anything else that happens that might be undesirable is governed entirely by what we call the Quality of Implementation (QoI), and is left for the clients and implementers of the specification to sort out. We will return to more valuable supererogatory behavior (with respect to contract checking) later in this paper.

Even a standard function, such `std::vector::size() const`, that faithfully delivers on its *essential behavior* yet logs each invocation to `std::err` — albeit an especially poor QoI, to say the least — would still be considered a *conforming implementation*. By contrast, if that same function ever threw the result, rather than returning it normally, that behavior would obviously no longer be true to its *essential behavior* and, hence, would *not* be considered conforming.

In short, we can and are expected to specify explicitly, in a function contract, the *essential behavior*, i.e., what *must* happen assuming all preconditions are

satisfied, but we simply cannot specify all the things that must *not* happen, leaving what must *not* happen to QoI, which seems to work out just fine in practice.

### 3.2 Implementation Failure

Sometimes a contract can be so demanding that it cannot be implemented perfectly or completely, at least not on the first version. Hence, the implementation might be good but not yet strictly conforming. A cardinal rule of function-contract edict, however, is that, if a function is called *in contract* and, for whatever reason, the implementation cannot deliver on the *essential behavior* in its contract, *it does not return normally*.

For example, imagine I have been assigned to write a value-semantic type (VST),[3] `Rlang`, that is used to characterize an arbitrary regular language, `L`. The class is currently semiregular,[4] and you have been asked to provide an operator equal that runs in less than some implementation-defined (but finite) number of seconds:

```
bool operator==(Rlang& lhs, Rlang& rhs);
   // Return `true` if `lhs` represents the same regular language as `rhs`;
   // otherwise, return false within 600 wall seconds of being invoked.
```

As we might surmise, determining whether two (nonidentical) finite-state machines accept the same language is a hard problem, at least as hard any NP-complete problem.[5]

Hence, no known implementation is guaranteed to work for this contract in general. Still, we give it our best try (because that's what we're paid to do), and if our time runs out, we have to do something other than return a `bool`.

One option would be to call terminate(), another would be to throw, yet another would be to block or spin, and still another would be to long jump. None of these approaches are good, but all of them are better than silently returning either `true` or `false`. At a minimum, we should indicate in the contract what happens if the function fails, e.g., throws "up," calls `terminate()`, or (yuck!) returns `false` to indicate uncertainty that the values of `lhs` and `rhs` are the same.

The Standard is required to be complete in its specifications. That is, *all* essential behavior *must* be specified. In addition, if a function might not return

---

[3] A value-semantic type is one that represents a platonic value that is independent of its representation, e.g., `5`, `|||||` and `101b` are each representations of the integer value five; see [Lakos15a].

[4] A semiregular type has all of the syntactic operations of a *regular* type (e.g., `int`, `std::string`, `std::complex<T>`), except for the equality comparison operations.

[5] [Lakos15b]

normally, that too *must* be specified. In practice, however, we often simply don't bother specify all the things that can go wrong, especially when they are sufficiently unlikely.

For example, suppose we are writing a portable lock handle, `mylock`, for a system lock. On construction, the lock handle allocates a lock from the operating system. Unlike other resources, the chance that no locks will be available is essentially zero. Hence, we customarily omit that bit from the contract and simply check to see if the resource was allocated. We would then simply terminate on failure.

When failure is possible but only through complete memory exhaustion, we sometimes skip specifically stating that function can throw, especially if that is obvious (at least to most) from the contract.

For example, let's take another look at our `stableSoryOnX` function from Section 3.1:

```
void stableSortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`, preserving the
    // original relative order of objects of equal `x` value.
```

A stable sort can run in O[n*log n] provided it has ample memory; an in-place implementation is inherently more complex and requires O[n*log^2 n] time. If we forgot to mention that this algorithm must run O[n*log n], we might also forget (or not bother) to say that it throws `std::bad_alloc` if the entire process runs out of memory.

In short; not every contract is always as explicit and complete as specified in the Standard; such is life in the real world.

## 3.3 Preconditions and *Undefined Behavior*

A *precondition* is a requirement (expressed or implied) that limits the circumstances under which a contract can be considered binding. In particular, *every* precondition of a function must be satisfied for a caller to be entitled to rely on *any* of the *essential behavior* promised in that function's *contract*.

Let's consider the familiar function, `sqrt`, that has a single stated precondition that its argument, `x`, must be non-negative:

```
double sqrt(double x);
    // Return a result, `y`, such that `y` is non-negative and that minimizes
    // the value of |y * y - x|.  The behavior is undefined unless `0 <= x`.
```

As long as the caller supplies a non-negative argument to `sqrt`, we have no indication that this function will do anything but return a valid answer. Hence, its *essential behavior* is that, given a non-negative argument, it will always return a valid result. Its contract didn't mention throwing, so we know it doesn't throw — at least not when *in contract*.

What happens if, for whatever reason, we were to somehow call `sqrt` with a negative value? Literally no information in the contract tells us what might happen; the contract offers no requirements on that behavior whatsoever.

*Undefined behavior* is defined in the Standard to be behavior for which there are no requirements, i.e., literally none! That doesn't mean, however, that if we were to invoke `sqrt(-1)` that literally *anything* could happen, but it does mean that anything that one could do deliberately or even accidentally in `sqrt` would be fair game:

```
double sqrt(double x)
{
    if (x < 0) {
        while (1) std::system("rm -rf *.*;*);  // very poor QoI but conforming
    }
    return sqrt_imp(x);  // some reasonable square-root routine
}
```

No decent developer writes code like the above, but many do write code that helps to detect *out-of-contract* calls, i.e., those that violate one or more preconditions:

```
double sqrt(double x)
{
    assert(0 <= x);      // assert precondition
    return sqrt_imp(x);  // some reasonable square-root routine
}
```

Some developers might even write code that allows the client to actually do something if they get a precondition wrong:

```
double sqrt(double x)
{
    if (0 <= x) throw "Hey, I said non-negative!  Now what?!";
    return sqrt_imp(x);  // some reasonable square-root routine
}
```

Because undefined behavior has no requirements, what happens when a function is called *out of contract*, i.e., with one or more of its preconditions violated, has nothing whatsoever to do with the specification of the function, i.e., its contract, and everything to do with its implementation, which might — and ideally would be — different in different build modes.

Now that we're getting good at this, let's ask an interesting question: Are there any other tacit preconditions for calling this function? Enough stack space

remains on the computer to accommodate at least two function calls. Should that be part of the explicit contract? No, because it would have to be repeated on literally every non-inlineable function. Great, let's not be silly.

Suppose we pass in a NaN value? What should happen? Now we need our language lawyer hat. Is the wording such that the explicit precludes a NaN? Well, if we had said, "The behavior is undefined *unless* `x` is a non-negative integer," then that `x` cannot be a `NaN` is clearly a precondition since `NaN` stands for *not a number*; a `NaN` argument is not a negative integer and, hence, it explicitly results in *undefined behavior*.

On the other hand, had we worded the precondition just slightly differently, we would have an entirely different result: "The behavior is undefined if `x` is a negative integer." `NaN` is still not a negative integer, but now it's also not explicitly undefined behavior either. So what is it? It's *implementation-defined* behavior, which means it must do something, but what it does is *implementation defined*.

As users, we don't need to be a language lawyers to know that the *essential behavior* of this function says absolutely nothing about what should happen if we pass in a `NaN`. The sage advice is, whether its *implementation-defined behavior* or *undefined behavior*, it's not *essential behavior,* and therefore it's *bad behavior* to rely on since it might change at any time without notice.

Let's go back and take another quick look at the contract for one of our sort functions from Section 3.2:

```
void sortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`.
```

Does `sortOnX` state any preconditions? No. Do all assumed preconditions need to be stated explicitly? No.

As it turns out, an implicit assumption is made that nothing outside the bonds of the language needs to be restated in a function's contract. For example, it is implicit in every contract that an object passed into a function must have been constructed and must not have been destructed prior to the invocation of that function. That literally goes without saying:

```
int main()
{
    int a, b;
    return average(a, b);
}
```

Imagine we had to write a contact that covered the obvious:

```
void sortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`.  The behavior is
    // undefined if either `b` or `e` have indeterminate value.
```

By the same token, imagine that every time we passed a pointer to an object to be modified in place, we had to state that the pointer was not null?

```
void sortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`.  The behavior is
    // undefined unless `b` and `e` are (1) not null and (2) refer to
    // a semiopen sequence of valid objects such that `e` is reachable
    // from `b`.
```

The burden of documenting what happens when a pointer is null is explicitly *not* a reason to prefer modifiable references to pointers or iterators. Rather, readers have the opportunity to ask themselves, “What is the explicitly stated *essential behavior* that will obviously occur when I pass in a null pointer for either `b` or `e`?” Since no obvious answer presents itself, assuming such *unspecified behavior* is even *implementation defined behavior* (as opposed to) *undefined behavior* would be the caller's fault. Note that every organization, development team, and library specification would be well advised to document what implicit assumptions are made for function preconditions in that context.

Finally note that the scope of preconditions is not limited to just the arguments of a function and may apply to literally anything, regardless of whether the function is capable of detecting whether the precondition is violated. Typical function preconditions, however, are limited to the arguments and any accessible (e.g., object or global) program state:

```
template <class T>
const T& Vector::front() const;
   // Return a reference to the nonmodifiable element at index position 0.
   // The behavior is undefined unless this object is not empty.

const T& Vector::operator[](std::size_t index) const;
   // Return a reference to the nonmodifiable element at index position 0.
   // The behavior is undefined unless `index < size()`.
```

The first member function, `front`, above takes no arguments; its only precondition is that the `Vector` object on which it is invoked is not currently empty. The second member function, `operator[]`, takes a single unsigned integer argument; whether the precondition is satisfied depends on both the value of that argument and the current state of the object. One could imagine preconditions that depended on global program state and perhaps even state external to the program, but that would be unusual and most likely not applicable to a general-purpose library, such as the Standard library.

## 3.4 Library UB versus Language UB

Recall that *undefined behavior* is simply the lack of any requirements whatsoever and, as currently defined in the Standard, applies to both language and library preconditions alike. That being said, for any library but the Standard Library (and only because it is the Standard[6]), an important *inherent* distinction exists between calling a library function *out of contract* and invoking a C++ language construct in a way that fails to meet its requirements.

Let's take another look at our `sort` function from the previous two sections:

```
void sortOnX(Point *b, Point *e);
    // Sort the specified `[b, e)` range of points in ascending order
    // based solely on their first coordinate, `x`.
```

Although not stated explicitly, we can deduce that no *essential behavior* is associated with passing in a null pointer for either `b` or `e`, so, as implementers, we are free to assign whatever *implementation defined* meaning we like:

```
void sortOnX(Point *b, Point *e) {
    if (!b || !e) throw "logic error!";
    sortOnX_imp(b, e);
}
```

There is nothing undefined about passing a null pointer to the conforming implementation of the `sortOnX` function as defined above. If either `b` or `e` (or both) is null, the function will throw in every build mode. Yet, according to its implied contract, no *essential behavior* is associated with null inputs. When we invoke a (nonstandard) library function for which one or more preconditions for that function are violated, we say that the function has *library* UB, irrespective of its implementation.

The compiler cannot yet read English, so it doesn't know when we have executed *library* UB. The compiler is, however, often aware when we have done something that would fail to satisfy the constraints of a language (or possibly Standard Library) construct at run time, and in those cases it is authorized to assume that such behavior will never execute and optimize accordingly. We refer to such classical *undefined behavior* as *language undefined behavior* or *language* UB.

Let's now take a look at another conforming implementation of `SortOnX;` this one prints a debug message when the list isn't empty and the first and last `Point` elements have unequal `x` values:

---

[6] Because the C++ language is closely collaborative with its Standard Library, certain functions in that library, such as `std::memmove`, are known to the language, and, thus, calling one of those *out of contract* might be considered tantamount to invoking a primitive language construct *out of contract*. Given a need, however, we could easily address those few special cases in the language to allow the distinction between *library*- and *language*-induced *undefined behavior* to apply equally to the C++ Standard Library as well.

```
void sortOnX(Point *b, Point *e) {
    if (b != e && b->x == (e-1)->x) std::cout << "sortOnX: unequal x values\n";
    sortOnX_imp(b, e);
}
```

In this implementation, supplying a null pointer for ether `b` or `e` (but not both) will require both `b` and `e` to be used in such a way that violates the requirement that null pointer values must not be dereferenced. Hence, invoking this implementation as in `SortOnX(0, a+3)` will cause the null value of `b` to be dereferenced, typically halting the program. In this scenario, we say that invoking *library* UB has led to *language* UB, whereas, in the previous example, no such language *UB* would have occurred.

To elucidate the important difference between these two implementations, let's create a small test program:

```
static Point a[] = { { 9, 1 }, { 9, 2 }, { 8, 3 } };

int main() {
    try {
        sortOnX(a, a + 3);
        std::cout << "First try worked!" << std::endl;
        sortOnX(0, a + 3);
        std::cout << "Second try worked!" << std::endl;
    }
    catch (const char *s) {
        std::cout << s << std::endl;
    }
}
```

If we plug in the first implementation above, the program would complete normally:

```
First try worked!
logic error!
```

In the example above, the first call worked normally, and the second call, which was absolutely *library* UB, terminated with a deliberately thrown exception. Even though *library* UB occurred, it didn't lead to any *language* UB. In other words, as far as the C++ *language* is concerned, this is a well-formed program and, when run, executes no (classical) *undefined behavior*.

Now, if we were to link with the second implementation and rerun the program, again we would almost certainly get a much different result:

```
First try worked!
Segmentation fault (core dumped)
```

This time, the first call to the function was fine, but the second call banged into *language* UB, and the program crashed.

The important takeaway from linking with these two different implementations of the same `SortOnX` and contract is that, regardless of which implementation we link to, the program called the `SortOnX` function *out of contract* and thus invoked *library* UB. In the first case, that library *UB* did *not* lead to *any* classic (language) UB, so the program was well behaved. When linking to the second implementation, however, the *library* UB caused a null-pointer to be dereferenced, which resulted in *language* UB. Unlike *library* UB, once language UB occurs, recovery is not guaranteed since the program itself is in an unknown state.[7]

Sometimes, whether library UB leads to language UB depends on the build mode. For example, let's look at another implementation:

```
void sortOnX(Point *b, Point *e) {
#ifndef NDEBUG
    if (!b || !e) throw "logic error!";
#endif
    sortOnX_imp(b, e);
}
```

If the implementation above is built normally, it will detect an *out-of-contract* call if either argument is null and will reliably throw `"logic error!"` If, however, the same implementation were built with the `-DNDEBUG` switch, the result of passing a null pointer as either argument would be left to the implementation of `sorOnX_imp(b, e)`. Although not explicitly stated, it would also be *library* UB and, almost certainly for any implementation, *language* UB to pass in two addresses where `e` was not reachable from `b`:

```
static Point a[] = { { 9, 1 }, { 9, 2 }, { 8, 3 } };  // one block of memory
static Point b[] = { { 9, 1 }, { 9, 2 }, { 8, 3 } };  // a separate block

int main() { sortOnX(a, b + 3); return 0; }  // almost certainly language UB
```

The program above will call `sortOnX` such that `e` is not after `b` in the same block of memory, which is (implicitly) *library* UB for this function. Many other implicit forms of *library* UB can also occur, such passing in improperly constructed (e.g., unaligned) arguments.

### 3.5 *Narrow* versus *Wide* Contracts

In the course of discussing contracts over the past two decades, a natural, inherent, and very important dichotomy has become clear: contracts that have

[7] Note that if the compiler can determine that a particular path within a program will necessarily lead to *language* UB, the optimizer is entirely within its rights to delete every instruction along that path back to the first branch where the flow of control might have chosen that direction and *assume* that the input will not go in that direction. Hence, any input that would have taken that path will likely not behave as the program author intended. This form of optimization, known as *time travel*, is one of the more insidious forms of bugs encountered in C++ programming.

preconditions and those that do not. Roughly a decade ago, I coined the terms *narrow* and *wide*, respectively, to characterize such contracts.[8]

A *narrow contract* has preconditions; a *wide contract* has none. A *narrow contract* admits *library* UB; a *wide contract* doesn't. Hence, the only way one can cause *language* UB as a result of calling a function having a wide contract is to call it outside the bounds of the language, such as passing in improperly formed objects (or, of course, if *language* UB has already occurred somewhere else in the program).

An observation that will become important shortly is that, for a function to have a *narrow contract*, there must be at least one combination of input and state values for which the behavior is *undefined*, such that invoking that function on those values would lead to *library* UB. Any contractual requirement or constraint placed on the behavior of the function when called with that input/state combination would mean that, *by definition*, that behavior is no longer *undefined behavior*.

Consider a strange function, `pos`, that returns its argument if positive:

```
int pos(int x);
    // Return `x` if positive.
```

Does this function have a *narrow* or *wide* contract? Again, we have to put on our language-lawyer hat and ask ourselves two questions: What are the expressed or implied preconditions, and what is the explicit *essential behavior*?

First, no explicit preconditions are stated. In fact, the only precondition is that the argument passed must not have indeterminate value:

```
int main() {
    int x;
    return pos(x);  // will result in library (and certain language) UB
}
```

The signature of the function tells us that it returns some value. The essential behavior is explicitly that if x is positive, the value of `x` is returned. The client can expect the function will always return, but it cannot expect anything about the value for values of `x` less than 1. Hence, the contract as stated is *wide*.

Had we instead explicitly stated the precondition that `x` be positive, then even the requirement to return is lifted for nonpositive values of `x`:

[8] Much more recently (i.e., May 9, 2023), we discovered, during the design of C++26 Contracts, the convenience of using the term *wide* to characterize a conforming implementation that would naturally accommodate a *wide* contract, even though the contract might have been explicitly defined as *narrow*.

```
int pos(int x);
    // Return `x` if positive; otherwise, the behavior is undefined.
```

The contract for `pos`, as amended, is now narrow, because now no requirements whatsoever are made regarding the behavior of `pos` for values of `x` less than 1.

### 3.6 `noexcept` and Contracts

Until now, everything we have discussed applies generally from C++98 through C++23 and beyond. As of C++11, however the `noexcept` specifier was invented to indicate programmatically to the compiler and to the human reader that a function so decorated will not allow an exception to escape from that function. If, at run time, an exception attempts to escape from such a nonthrowing exception specification barrier, it will be caught by the C++ runtime and `std::terminate()` will be invoked unconditionally.

As a pedagogical experiment, let's revisit our `average` function from Section 3.1:

```
int average(int x, int y);
    // Return the closest integer, `z`, to the mean of `x` and `y`; if `z` is
    // not unique, return the larger one.
```

First, we observe that this (pure) function has no stated or implicit preconditions; hence, it has a wide contract. Second, we see that its essential behavior requires it always to return normally; hence, it doesn't, for example, abort, terminate, long jump, throw, block, or spin indefinitely.

Suppose we were to add a clause to the contract such as "`does not throw`" or, as is commonly done in the C++ Standard today, "`throws nothing`." Apart from restating the obvious, this clause would not in any way change the clear and incontrovertible meaning of the contract.

Now suppose we were to add `noexcept` to the declaration of an otherwise identical function, `averageNE`, and leave the contract unchanged:

```
int averageNE(int x, int y) noexcept;  // nonthrowing exception specification
    // Return the closest integer, `z`, to the mean of `x` and `y`; if `z` is
    // not unique, return the larger one.
```

The contract for `averageNE` is the same as for `average`, but now the compiler knows that this new function is not allowed to throw and hence will *not* lay down code to guard against an uncaught exception escaping from the function at run time:

```
void g1()          {             }  // may throw but doesn't
void g2() noexcept { throw "up"; }  // may not throw but tries to anyway
                                    // If `g2()` is called, program terminates.
template <typename F>
```

```
constexpr isNoexcept(F f)
// Return true if `f` may throw; return false otherwise.
{
    return noexcept(f());  // `noexcept` operator applied to invocations
}

static_assert(false == isNoexcept(g1));  // `g1` has    throwing specification.
static_assert( true == isNoexcept(g2));  // `g2` has nonthrowing specification.
```

As the code snippet above illustrates, the `noexcept` operator is oblivious to the function's implementation; it reports back only what the declaration promises.

In short, adding `noexcept` to a function whose wide contract already implies that its essential behavior requires it to return normally has no effect on its binding contract but may have other unintended collateral effects (see below).

As a second example of the effects of applying `noexcept` to a function having a wide contract, let's consider a widely used, application-specific sort routine, `businessSort`, that is used widely throughout our company:

```
void businessSort(Record *b, Record *e);
    // Sort the specified `[b, e)` range of `Record` objects in
    // ascending order of primary fields.
    // The runtime complexity is O(N *log N).
```

As of now, this function does not need to acquire any additional resource to perform its task and is therefore duty bound by its essential behavior to return without fail; i.e., it proports to be a *nofail function*.[9] Hence, we would be surprised if this function were to throw an exception. Still, the contract doesn't explicitly say that it doesn't, and if were someday to need to allocate a temporary resource (e.g., dynamic memory) and that resource were not available, throwing an exception might be the most natural *and automatic* way of handling such a highly unlikely failure.[10]

If, for whatever reason, stating unequivocally that a function does not now and never will throw — at least not for the *current* set of preconditions — is deemed critically important, then adding that promise to the essential behavior — e.g., either "`does not throw`"or "`throws: nothing`" — is all that is needed. Now we have flexibility; if this function were to throw given the current set of valid inputs, it would be grossly nonconforming:

```
void businessSort(Record *b, Record *e);
    // Sort the specified `[b, e)` range of `Record` objects in
```

[9] [Lakos22a], Section 3.1"`noexcept` Specifier," "Potential Pitfalls," "`noexcept` versus nofail," pp. 1122–1123

[10] Another alternative might be to return status; however, all existing uses of the function would fail to check the status, and all future uses would be burdened with having to check it even though the likelihood of system-wide memory exhaustion might be vanishingly small.

```
    // ascending order of primary fields.
    // The runtime complexity is O(N *log N). Throws: nothing.
```

When a function has a *wide* contract, it has no preconditions. Thus, when we say " `does not throw (in contract)`," we mean that the function doesn't throw at all. Another way to communicate that same contractual obligation for a function that already has a wide contract is to decorate the function with the `noexcept` specifier:

```
void businessSort(Record *b, Record *e) noexcept;  // `noexcept` here implies
                                                   // "Throws: nothing."
    // Sort the specified `[b, e)` range of `Record` objects in
    // ascending order of primary fields.
    // The runtime complexity is O(N *log N).
```

At this point, we are about to make a bold statement that might seem a bit shocking to some but, as will be proven handily in subsequent sections using *Liskov Substitutability*, is manifestly true. To remove emotion from this demonstration, let's change gears and talk about another aspect of C++, namely functions that don't return.

For example, consider a function, `handler`, that prints its positive argument and then terminates the program().

```
void handler (int x)                                          // Version A1.0
    // Print the value of `x` to standard out and then call `std::terminate()`.
    // The behavior is undefined unless `1 <= x`.
```

This function clearly has a narrow contract whose domain is defined to be positive values of `x`. If we call this function with, say, the value 5, we know that it will print 5 and terminate. If we were to later replace this function with another one that, say, threw `std::logic_error` on an input of zero, that would be a backward compatible change because any programs written to the old standard will continue to work:

```
void handler (int x)                                          // Version A2.0
    // If  `x` is positive, print the value of `x` to standard out and then call
    // `std::terminate()`; otherwise just throw `std::logic_error`. The behavior
    // is undefined unless `0 <= x`.
```

Notice that all programs written to Version A1.0 continue to work *because* the contract is *narrow*, and *no* requirements are placed on the behavior of calling this function with values less than 1. Now suppose we want to extend this function's contract again (in a backward compatible way, of course) such that it is now even wider:

```
void handler (int x)                                          // Version A3.0
    // If  `x` is positive, print the value of `x` to standard out and then call
    // `std::terminate()`; otherwise , if `x` is 0, just throw `std::logic_error`;
    // something will print (not saying what) and then return.
```

Now we have a wide contract because we know that the function must terminate on positive `x`, throw when `x` is zero, and return when `x` is negative. There is no combination of input and state values for which there are no requirements on `handler`, so version A3.0 of `handler` now has a (backward-compatible) *wide contract*.

Now suppose that on our original version we, instead of writing out that it didn't return, had decided to document it another way, namely in code using the `[[noreturn]]` attribute.

```
[[noreturn]] void handler (int x)                                    // Version B1.0
    // Print the value of `x` to standard out and then call `std::terminate()`.
    // The behavior is undefined unless `1 <= x`.
```

According to the documentation, this function appears to have a narrow contract. But that's simply not the case. There is an irrevocable and permanent requirement on every combination state and input for this function — and all future versions of this function — that it doesn't throw. It's as if we had said in the verbal contract

```
[[noreturn]] void handler (int x)                                    // Version B1.0
    // If `x` is positive, print the value of `x` to standard out and then call
    // `std::terminate()`; otherwise, (maybe do something) and then return.
```

Any future backward-compatible version of this function be unable to violate this basic contract. Consider that Version A2.0 is a viable next version of this contract, but Version A3.0 is not. Version A3.0 is not a backward-compatible version because the original B1.0 version of the contract was *wide* and had a behavior requirement for negative values that contradicted what we wanted to do in A3.0. Hence, just by adding the `[[noreturn]]` attribute to the declaration of the `handler` function, we turned its *narrow* contract into a *wide* one that blocks us from creating the otherwise backward-compatible enhancement we wanted.

Let's now consider a function, such as `sqrt`, that has a *narrow contact* that explicitly guarantees not to throw *in contact*:

```
double sqrt(double x);
    // Return a result, `y`, such that `y` is non-negative and that minimizes
    // the value of `|y * y - x|`.  Does not throw (in contract).  The behavior is
    // undefined unless `0 <= x`.
```

Again, unlike a wide contract, the domain of a narrow contract does not contain all syntactically valid input/state combinations. Invoking such a function out of contract, according to the Standard today, results in undefined behavior. Recall that undefined behavior is defined in the Standard to mean behavior that has no requirements.

Just by decorating the sqrt function above with the `noexcept` specifier, we put an absolute requirement on every input/state combination. The implications for the new contract are significant:

```
double sqrt(double x) noexcept;
    // If `x` is nonnegative, return a result, `y`, such that `y` is non-negative
    // and that minimizes the value of `|y * y - x|`; otherwise can do anything
    // that is not undefined behavior except throw.  The exception specification
    // of this function is non-throwing as observed by the `noexcept` operator.
```

Not only can we no longer safely widen the contract to throw on a negative value, we cannot remove the `noexcept` operator as that too would be a non-backward-compatible change to the explicitly codified contract.

In short, by virtue of adding the `noexcept` specifier to a function having an otherwise narrow contract, we have placed an essential-behavior requirement on every possible syntactically valid state/input combination. There simply cannot be a function having both a *narrow contract* and a nonthrowing exception specification. Hence, narrow contracts and `noexcept` are inconsistent as well as being inherently incompatible practically.

# 4 THE NEED FOR NARROW CONTRACTS

Recall that a *narrow* contract is one that has preconditions, and thus at least one combination of input and state values exist for which absolutely no requirements are placed on the function's behavior — essential or otherwise. When a function is called *out of contract*, its behavior is entirely *undefined*. Wide contracts can be preferred for many reasons, such as end-user interfaces. In this section, we will explore some of the fundamental benefits of narrow contracts and why they are so incredibly important to the sound design of low-level C++ library software.

## 4.1 Cost-Effective Design, Development, Testing, and Performance

Narrow contracts are a mainstay of effective software design, and especially so in languages, such as C and C++, where code size (particularly on the hot path) and runtime performance are often at a premium. Compared to wide contracts, narrow contracts offer many practical advantages. To get us started, let's consider an implementation of a factorial function, `fact`, that takes an integer and returns an integer:

```
double fact(int n);
    // Return result of 1.0 multiplied, in turn, by each integer in the
    // closed range [1..n].
```

The contract above is wide because the defined behavior applies to all syntactically valid integers, namely that for all `n` less than 2, `fact(n)` is exactly 1. It almost seems as though someone looked at its implementation and then documented it:

```
double fact(int n)
{
    double r = 1.0;
    while (n > 1) r *= n--;
}
```

Given this wide contract, it's hard to imagine any implementation that performs better. Generally speaking, however, a narrow contract will often be faster than a wide one because a narrow contract doesn't need to check the internal boundaries of the algorithm. For example, consider a function, `mySqrt`, that is given a wide contract:

```
double mySqrt(double value);
    // Return the positive square root of the specified `value` if
    // `0 <= value`, and 0.0 otherwise.
```

The implementation of such a function is easy to imagine:

```
double mySqrt(double value)
{
    return 0 <= value ? std::sqrt(value) : 0;
}
```

Now consider what would happen if we instead made the contract for `mySqrt` narrow:

```
double mySqrt(double value);
    // Return the positive square root of the specified `value`.  The behavior
    // is undefined unless `0 <= value`.
```

With this contract, the previous implementation still works just fine, but now we have another, presumably faster (but no slower) implementation:

```
double mySqrt(double value)
{
    return std::sqrt(value);
}
```

By making the contract narrow, we have transferred the checking requirement to the client who may or may not already know that the value is appropriate for the call. If the client doesn't know, we've lost nothing, but if the client does know, we just eliminated a branch.

Performance is only the first of many reasons why narrow contracts are practically useful. Software developers in industry are often responding to a business need. Sometimes that need isn't immediately fully baked, and we need to solve the part that we understand as quickly as possible without precluding future (backward-compatible) enhancements (as discussed at length in subsequent sections). Imagine we are to write a gaming function that takes a pair of integer coordinates, `x`, `y`, and does something with them. Our management hasn't decided what happens if either is negative, but they're

working on it. Meanwhile we need to quickly complete the part they've specified:

```
double doSomething(int x, int y)
    // Something shall be done!
    // The behavior is undefined unless `0 <= x` and `0 <= y`
```

To get the work started while also preserving our options, we have decided to implement the part we understand today and to keep open the option of widening our domain if and when we figure out what we need to do.

Let's take a quick look back at our contract for our factorial function, `fact`:

```
double fact(int n);
    // Return result of 1.0 multiplied, in turn, by each integer in the
    // closed range [1..n].
```

Because we defined it to be wide, we have no ability to do more. Had we instead made it narrow, i.e., defined only for integer values greater than or equal to zero, we would then be able to extend the domain to include nonintegral positive and negative values (as defined by the gamma function) without impacting any existing clients:

```
double fact(double n);
    // Return the gamma function applied to `n`.  The behavior is undefined
    // unless `0 <= n` or `n` is not integral.
```

From a cost/benefit perspective, narrow contracts are superior. Even if we know what to do, if no one needs it, why implement it. If we implement it, it has to be

- designed
- documented
- coded
- tested
- maintained

As a rule, less code (on the hot path) runs faster, is cheaper to implement and maintain, and keeps our options open for future backward-compatible enhancements. Except for interfacing with (unsophisticated) end users, when it comes to narrow contracts, what's not to like?

## 4.2 Design by Contract (DbC) and the Liskov Substitution Principle (LSP)

When it comes to designing contracts, both in general and those related by subtyping via inheritance and virtual functions, the classic advice comes from

Bertrand Meyer[11] in what he calls *Design by Contract*.[12] In this paradigm, every function is expressed in terms of preconditions and postconditions, such that the valid domain (preconditions) of every base-class function, `B::f`, is expected to be a subset of any corresponding derived-class function, `D::f`, and vice versa for the range (postconditions):

```
(D1) (D2) (D3)       int D1::f(int i) override;  // pre: 0 <  i   post: 0 <  i
    \  |   /         int D2::f(int i) override;  // pre: 0 <= i   post; 0 >= i
     v V  v          int D3::f(int i) override;  // pre  true     post: 0
     ( B  )  virtual int  B::f(int i);          // pre: 0 <  i   post: true
```

As the example above illustrates, the base class, `B`, has a virtual function, `B::f`, which has a narrow contract that accepts any positive value of its argument, `i`, and may return any integer value. The derived class, `D1`, overrides `B::f` with a function, `D1::f`, that also has a narrow contract that accepts any positive integer, but this function promises to return only positive values of `i`. `D2` also overrides `B::f` but with a function, `D2::f`, that accepts any non-negative integer and promises to return only nonpositive ones. Finally, `D3` too overrides `B::f` but with a function, `D3::f`, having a wide contract that always returns 0.

Importantly, this design methodology is focused entirely on telescoping subsets of preconditions and postconditions for virtual functions in inheritance hierarchies whose raison d'être is to allow for *variation in behavior*.[13] For example, the classic object-oriented (OO) system that draws a shape depending on its runtime type clearly executes different behavior depending on its derived type, even if all of the domains and ranges of the hierarchy conform to DbC:

```
(Rectangle) (Circle) (Polygon)     void Rectangle::draw() const override;
            \   |    /             void    Circle::draw() const override;
             v  V   v              void   Polygon::draw() const override;
               (Shape)      virtual void     Shape::draw() const override;
```

Something that is specifically *not* addressed by DbC is the rarely assumed requirement that the behavior of the derived class function behave as if the base-class function had been called for all valid inputs of the base class. For example, let's consider a base-class `Bool` that has a single function, `g` that takes a `bool`, `b`, and returns that `bool`:

```
bool Bool::g(bool b) { return b; }
```

---

[11] Fun Fact: My first course in object-oriented design (c. 1988) was with Bertrand Meyer. I specifically recall asking him how one would write an object-oriented program that modeled comparing apples and oranges in pre-Standard C++. I never got a straight answer, but I figured it out on my own (well before there were dynamic casts) using local static variable addresses as type ids.

[12] Meyer applied to trademark "Design by Contract (DbC)" in 2003; the trademark was granted in 2004.

[13] [Cargill92]

Now, according to DbC, exactly three other pure functions that can be derived from `Bool`:

```
(True) (False) (Fool)  void  True::g(bool b) const override { return true;  }
       \   |   /       void False::g(bool b) const override { return false; }
        v  V  v        void  Fool::g(bool b) const override { return !b;    }
         (Bool)virtual void Shape::g(bool b) const          { return b;     }
```

Each of the functions above satisfies the classic DbC requirements that the preconditions of `g` for the derived classes are *no narrower* than they are for the base-class `g` and the postconditions for the derived `g` member functions are *no wider* than they are for the base `g`:

```
     True::g      False::g     Fool::g          _______RESULT________
    pre: [0 1]   Pre: [0 1]   pre: [0 1]        Member   <- INPUT ->
   Post: [_ 1]  Post: [0 _]  post: [0 1]       Function   true false
           \       |       /                   --------  ----- -----
            v      v      v                     True::g   true  true
              pre: [0 1]                       False::g  false false
             post: [0 1]                        Fool::g  false  true
               Bool::g                          Bool::g   true false
```

That defines a kind of substitution principle that allows for variation in behavior, which is exactly what virtual functions were designed to do. Note it does *not* imply that if you pass a derived class into a function that takes a pointer or reference to the base class, you will necessarily get the same behavior as you would have had you passed in the base class (or any other derived class for that matter):

```
void h() {
    assert( true == f( Bool(), true));  assert(false == f( Bool(), false));
    assert( true == f( True(), true));  assert( true == f( True(), false));
    assert(false == f(False(), true));  assert(false == f(False(), false));
    assert(false == f( Fool(), true));  assert( true == f( Fool(), false));
}
```

Although this much stronger substitution property is not relevant when dealing with *interface* and *implementation* inheritance (each of which involves virtual functions), it does pertain to sound *structural inheritance*; i.e., inheritance in which nonvirtual functions *hide* other nonvirtual functions in some direct or indirect base class, something many popular books advise against.

Hiding functions (as opposed to overriding them) is generally ill advised because it makes the behavior of the function dependent on the static type from which it is called. Imagine, in the example above, that we had instead chosen to make the `g` function nonvirtual. Had we done that, we would have introduced the typically undesirable property known as *slicing*, in which passing an object by reference implicitly reverts all of its nonvirtual functions to their base-class contracts:

```
void h2() {
    assert( true == f( Bool(), true));  assert(false == f( Bool(), false));
```

```
    assert( true == f( True(), true));  assert( true != f( True(), false));
    assert(false != f(False(), true));  assert(false == f(False(), false));
    assert(false != f( Fool(), true));  assert( true != f( Fool(), false));
}
```

The asserts in `h2()` above communicate the slicing behavior that would occur if `g` were not declared virtual in `Bool`. But consider a property that, if a derived class is sliced down to its base class, behaves *as if* it were the base class for all programs written in terms of that base class. This strong notion of supertype and subtype is the heart of the *Liskov Substitution Principle* (LSP). Contrary to popular belief, LSP has absolutely nothing to do with any notion of sound design involving virtual functions.[14]

### 4.3 Structural Inheritance and *Safe* Substitutability

At the highest level, the seminal property that LSP identifies with respect to types is that a *Liskov Substitutive* subtype `S` of a supertype `T` can be used in any programming context where `T` can be used. That is, if we were to replace every *object* of type `T` with an object of type `S`, the observable behavior of the program would be *unchanged*. Put another way, an `S` behaves like and is as good as a `T` for every situation in which a `T` can be used and presumably for a few other programming contexts as well.[15]

As an example of *structural inheritance* conforming to LSP, consider the `const` member function `operator[]` on the standard container, `std::vector`:

```
template <class T>
const T& vector::operator[](std::size_t index) const
    Return the element at the specified index.  The behavior is undefined
    unless `index < size()`.  Throws nothing.
```

The essential behavior is explicit in that this function doesn't throw — that is, not when invoked *in contract*. Calling this function with an index that is beyond the end of the sequence, however, would be (at least) *library* undefined behavior, and throwing would, of course, not be precluded.

---

[14] Fun Fact: Robert C. Martin, “Uncle Bob,” and I were having dinner one night after he spoke at Bloomberg, and we were discussing what people meant by the LSP. Not too long after, I invited Dr. Barbara Liskov to speak at Bloomberg about her abstraction work, after which I took her out for dinner. I asked her if her substitution principle had anything to do with virtual functions. Her answer was, "No." After that, I asked her if she would mind signing, for my daughter Sarah (a CS major), one of the programs I had made for her OOPSL keynote talk. She agreed. Then I asked her to sign one for me, and she graciously did. Finally, I asked her to sign one more to Uncle Bob. She asked, "What should it say?" I replied, "Dear Uncle Bob, John was right." I then sent the signed program to Mr. Martin along with the clip from Annie Hall where the know-it-all Columbia adjunct professor (I was one from 1991–1997) is standing on a movie line pontificating to Annie about what Marshall McLuhan's writing means, so Woody Allen, disgusted, steps off screen and brings Mr. McLuhan back live to speak for himself! (See https://www.youtube.com/watch?v=9wWUc8BZgWE.) Upon receipt, dear Uncle Bob, in his typical regal eloquence, quipped, "I've been *poned* [powerfully owned]."

[15] [Liskov87]

Out-of-bounds errors are among the most notorious and frequent defects, especially for student programmers. While at Texas A&M, one of Professor Stroustrup's exercises for students was to implement what he calls a `CheckedVector`. A `CheckedVector` is everything that a `std::vector` is except the bracket operator has a *wide* contract. A straightforward way to implement the `CheckedVector` uses structural inheritance and, in C++11 or later, inheriting constructors (otherwise we would need to forward them by hand)[16]:

```
template<class T>
struct CheckedVector : std::vector<T>
{
    using std::vector<T>::vector;  // inheriting constructors
    T& operator[](std::size_t index) {
        if (index >= this->size()) throw std::range_error("bad index");
        return std::vector<T>::operator[](index);
    }
    const T& operator[](std::size_t index) const {
        if (index >= this->size()) throw std::range_error("bad index");
        return std::vector<T>::operator[](index);
    }
```

Now if the student happens to access a `CheckedVector` out of contract, it will throw, yet an object of this type can be used in every context in which an `std::vector` can be used:

```
double average(const std::vector<int>& sequence);
    // Return the average of the elements in the specified `sequence`.
    // The behavior is undefined if `sequence` is empty.

int main() try {
      std::Vector<int>  vec {1, 2, 3, 4, 5};
    CheckedVector<int> cvec {1, 2, 3, 4, 5};

    double  d = averate(cvec);
    double cd = average(cvec);  assert(cd == d);

    int  i = vec[2]
    int ci = vec[2]  assert(ci == i);

    int  j = vec[5];  // undefined behavior
    int cj = vec[5];  // well defined: throws `std::range_error`

} catch (const std::exception& e) { std::cout << "Oops! " << e.what(); }
```

We say that the `CheckedVector` is *Liskov Substitutable* for `std::vector` because, in every defined use of `std::vector`, replacing an `std::vector<T>` object with an object of type `CheckedVector<T>` will have precisely the same observable behavior. In other, new situations, however, calling the bracket operator will become defined behavior, throwing an `std::range_error`

[16] To minimize clutter, we are eliding the optional `allocator` parameter.

exception, rather than the entirely unspecified *undefined behavior* for `std::vector`.

Note that `std::vector` follows the Lakos Rule in that it doesn't have both a *narrow contract* and a nonthrowing exception specification. Before moving on to even more compelling reasons why the Lakos Rule is important for the C++ Standard Library, let's consider what the consequences to `CheckedVector` would be with regard to Liskov Substitutability if, say, in C++26 we were to break the Lakos Rule and put `noexcept` on the two `std::vector::operator[]` overloads.

Academically, it would mean that a `CheckedVector` would no longer *Liskov Substitutable* for an `std::vector`. Placing `noexcept` on the declaration places an explicit requirement on the essential behavior for every input/state combination, namely that this function shall not throw. The contract is thus wide, and to be *Liskov Substitutable* the one thing that a *checked vector* must not do is throw. In fact, even applying the weaker requirement of DbC, `CheckedVector` fails the test because the range of behaviors for `std::vector`'s bracket operator is now narrower than the range for that of `CheckedVector`.

As a matter of coding, it's trivial to exhibit situations in which an expression involving a `CheckedVector` would yield radically different code paths from one that involved an `std::vector` were it to violate the Lakos Rule and have a `noexcept` bracket operator:

```
template <class C>
void poke(C& container, std::size_t index)
    // Poke the element at the specified `index` of the specified `container`.
    // The behavior is undefined if index is not in range.
{
    if constexpr ( noexcept(container[0]) ) {  // fast algorithm
        // ... Do something quick and dirty.
    }
    else {                                     // slow algorithm
    {
        // ... Do something else slow and neat.
    }
}
```

Regardless of anything else, if the bracket operator of the container passed (as the first argument) to the generic function `poke` has a *throwing* exception specification, the *slow* algorithm will be instantiated regardless of anything else; otherwise, the *fast* will be instantiated. Given that these algorithms are entirely separate code paths with potentially entirely different observable behaviors, placing `noexcept` on `std::vector::operator[]`would eliminate an important design strategy such as this sound, safe, *Liskov Substitutable* structural inheritance.

Now the astute reader might ask, why would anyone choose to write a generic function that branched on the exception specification of the bracket operator of a container type, knowing full well that it would never throw in contract? The answer to that question is ideally no one, which is exactly why there is almost certainly no reason to decorate such an undeserving member function with `noexcept` (see "increasingly dubious reasons to use `noexcept`" below).

### 4.4 Contract Extension and Backward Compatibility

The examples above with structural inheritance were largely academic and not necessarily compelling to pragmatic, results-oriented coders who want to design code that is easy to maintain, support, and extend over time. Perhaps surprisingly, the notion of *Liskov Substitutability* applies much more powerfully to versioning.

Suppose we forget about inheritance completely and instead think of a subtype `S` as a subsequent version of the current unit of software `T`. The substitution principle we seek here is one that ensures backward compatibility. As it turns out, what we are looking for is again nothing other than *Liskov Substitutability* applied to atomic units of logical and physical design, e.g., `.h`/`.cpp` pairs, that we called *components*.[17,18]

Let's now recast LSP to apply more generally, as we're suggesting:

> For all current programs `P` built using component `C`, if we were to replace the source code of `C` with a newer version, `D`, and rebuild each `P`, there would be no observable difference in behavior in any `P`.

The principle is as powerful as it is simple and is the goal of every group maintaining, extending, and deploying library software.

As a first example, suppose we have some end-user API consisting of two free functions, `globalLookup` and `globalSize`, that provide access to a global store of values that can never be negative:

```
int globalSize();                                          // Version A1.0
    // Return the number values in the global store.

double globalLookup(int i);  // version 1.0
    // Return the value at index position `i`.  The behavior is undefined;
    // `0 <= i < globalSize()`.
```

[17] [Lakos20], Section 0.7, “Physically Uniform Software: The Component,” pp. 46–57

[18] This real-world, practical notion of substitutability — in addition to all logical behavior — may comprise many other dimensions of backward compatibility, such as run time, compile time, executable size, physical dependencies, testability, code quality, licensing, maintenance costs, and so on.

After the initial release, we determine that clients want to access the one-past-the-value, and when that happens, they would like the lookup function to return a negative value (as a flag).

Notice that the initial range of our `globalLookup` function, i.e., its postcondition, was to return a non-negative value. But now that we have widened the domain, the new inputs can do anything without affecting any previous clients:

```
int globalSize();                                                  // Version A2.0
    // Return the number values in the global store.

double globalLookup(int i);
    // Return the value at index position `i` unless `i == `globalSize()`,
    // in which case, return `-1.0`.  The behavior is undefined unless
    // `0 <= i < globalSize() + 1`.
```

Over time, out-of-contract calls by clients force another change, this time to a wide contract in which inputs that are either negative or greater than one past the end of the store cause an `std::range_error` to be thrown:

```
int globalSize();                                                  // Version A3.0
    // Return the number values in the global store.

double globalLookup(int i);  // version 2.0
    // If `0 <= i < globalSize()`, return the value at index position `i`;
    // otherwise, if i == globalSize()`, return `-1.0`; otherwise, throw
    // `std::range_error`.
```

Notice that this sub-subcomponent has a wider range then either of its predecessors, which is of no consequence because the new range values map onto what was undefined behavior in the previous version. Because each version is *Liskov Substitutable* for the previous one, a new version satisfies a wider audience without disenfranchising any of its current clients.

Consider now what would have happened had the author of the original version of the API decided why not just decorate both with `noexcept`, since neither of these functions can throw in contract:

```
int globalSize() noexcept;                                         // Version B1.0
    // Return the number values in the global store.

double globalLookup(int i) noexcept;  // version 1.0  (BAD IDEA)
    // Return the value at index position `i`.  The behavior is undefined;
    // `0 <= i < globalSze()`.
```

Because the first function has a wide contract, adding `noexcept` doesn't violate the Lakos Rule. On the other hand, adding `noexcept` to the second one, which is narrow, *does* violate the Lakos Rule. Now, as a consequence, the second version is still in scope but not the third. Hence, violating the Lakos Rule frequently interferes with making desirable backward-compatible enhancements, especially those that might not be immediately foreseeable.

Low-level functions are much easier to specify precisely. In practice, however, the further we go up into applications, the less sable and more malleable[19] contracts become. Just because something doesn't throw *in contract* today doesn't mean that it never will.

Let's imagine that we have a legacy application-level domain-specific sort function, `businessSort`, that is used widely throughout our organization. The algorithm is tuned to the current business needs, which means that the record keys on which it sorts are subject to modification and thus not explicitly specified in the contract:

```
struct Brec { /* ... */ };  // malleable business record      // Version A1.0

void businessSort(std::vector<Brec>& records)
    // Sort the elements in the specified `records` sequence in ascending
    // order according to the currently established primary fields in time
    // proportional to N*log N where N = `records.size()`.
```

Although not explicitly stated, the contract makes clear that there is no reason to throw because there is no need to allocate any additional resources to achieve the stated *essential behavior*. Because this is a business application, there are already many cases in which allocating memory could conceivable throw (e.g., due to memory *fragmentation* if not *exhaustion*). Thus, the application has established a safe-shutdown mechanism to address `std::bad_alloc` exceptions thrown from the global allocator.

Imagine now that customers don't like that their nonprimary fields are getting reordered when they invoke this sort, so management has decreed that the sort shall be stable. Recall that a stable sort can be executed in O[N*log N] time, provided that sufficient additional memory is available, otherwise, the best known in-place algorithm will take (worst case) O[N*log^2 N] time. We have hard limits on our response time, and if that is exceeded, the request will fail and the customer will be even more unhappy.

Given the constraints, a decision is made that we will try to allocate the memory and hope it proceeds, knowing that, specifically, an `std::bad_alloc` might have to be handled along with other such exceptions at a higher level:

```
struct Brec { /* ... */ };  // malleable business record      // Version A2.0

void businessSort(std::vector<Brec>& records);
    // Sort the elements in the specified `records` sequence in ascending
    // order according to the currently established primary fields, keeping
    // the relative order of all equivalent records stable in time
    // proportional to N*log N where N = `records.size()`.
```

[19] [Lakos20], Section 0.5, "Malleable vs. Stable Software," pp. 29–43

Notice that this function is technically *not Liskov Substitutable* for the other one. That said, the specific condition in which it is not is (1) exceptionally rare, (2) easy to understand, and (3) fits right into the already-established infrastructure to handle such exceptional conditions — specifically failure to allocate dynamic memory.

Now suppose instead we had decided on our first version that, because our `businessSort` function had no need to allocate today, we would go ahead and promise that it would never throw in any future version:

```
struct Brec { /* ... */ };  // malleable business record      // Version B1.0

void businessSort(std::vector<Brec>& records) noexcept;
    // Sort the elements in the specified `records` sequence in ascending
    // order according to the currently established primary fields in time
    // proportional to N*log N where N = `records.size()`.  Does not throw.
```

Because we have explicitly promised that it doesn't throw now or ever, we might choose to go ahead and decorate the function with `noexcept`. The affect is that, anyone who checks this function using the `noexcept` operator will learn that they can write generic functions that will take different code path, knowing that this specific function will not throw. The extending this contract to the one in version A2.0 is no longer viable while preserving any semblance of backward compatibility.

In short, one is well advised to consider — unless its already explicit in the contract — whether one wants to promise that even a *wide* contract, let alone a *narrow* one, that doesn't throw today never will.

### 4.5 Wide Implementations for Narrow Interfaces

A contract is an agreement between two parties for a minimum level of service, assuming all preconditions are met. We can think of any additional service as a free bonus, provided that all essential behavior stated in the contract on the interface is satisfied (a.k.a. *conforming*).

As a second gedankenexperiment, let's imagine that the implementation as written implies a second (English) contract whose preconditions are a superset (but whose postconditions are not necessarily subset) of the documented contract in the interface. Let's further model this hypothetical (implementation) contract as what is planned to become the public contract for the next version of the software. For an implementation to be conforming, its implied contract must be *Liskov Substitutable* for the contract associated with its interface.

For example, suppose we have a `Point` class that is designed to hold two *signed* integer coordinates `x` and `y`:

```
class Point {                                                 // Version A1.0
{
```

```
    int d_x;  // must hold values in range [-16384 .. 16384]
    int d_y;  //  "    "      "    "   "       "    "   "

  public:
    static bool check(int z);      // If `z` is out of range throw "Error!".
    Point(int x, int y) : d_x(x), d_y(y) { check(x); check(y); }
       // Create a point object having the respective `x` and `y` coordinates.
       // The behavior is undefined unless the absolute values of x` and
       // `y` are each no larger than 16384.

    int x() const { return d_x; }  // Return the current x-coordinate
    int y() const { return d_x; }  // Return the current y-coordinate
}
```

Despite the *wide* initial implementation, we expect that we will have too many points in our, e.g., ICCAD, system to waste over half the space, so we plan to reimplement the data members as short `int` in the next version:

```
class Point {                                                  // Version A2.0
{
    short d_x;  // must hold values in range [-16384 .. 16384]
    short d_y;  //  "    "      "    "   "       "    "   "
                        *no changes below this line*
```

Had we instead initially left the contract naturally wide, we would not have been able to supply the new implementation in a *Liskov Substitutable* way, and thus many existing clients might be forced to rework their code.[20]

Let's now consider the classic application of having an implementation whose implied contract is wide yet conforms to the narrow contract of its published interface. Rather than create an example, let me instead reprise a page from legendary C++ researcher, author, and trainer, Scott Meyers' final book, *Modern Effective C++*[21]:

> It's worth noting that some library interface designers distinguish functions with *wide contracts* from those with *narrow contracts*. A function with a wide contract has no preconditions. Such a function may be called regardless of the state of the program, and it imposes no constraints on the arguments that callers pass it.[22] Functions with wide contracts never exhibit undefined behavior.

---

[20] An example can be found in [Lakos22a], Section 3.1."`final`," "Use Cases," "Suppressing derivation to ensure portability," pp. 1014–1015. Note that, prior to C++11, we might well have relied on just a narrow contract, rather than implementing some horrible kluge, e.g., involving virtual inheritance.

[21] [Meyers15], Item 14, pp. 95–96

[22] "'Regardless of the state of the program' and 'no constraints' doesn't legitimize programs whose behavior is already undefined. For example, `std::vector::size` has a wide contract, but that doesn't require that it behave reasonably if you invoke it on a random chunk of memory that you've cast to a `std::vector`. The result of the cast is undefined, so there are no behavioral guarantees for the program containing the cast."

Functions without wide contracts have narrow contracts. For such functions, if a precondition is violated, results are undefined.

If you're writing a function with a wide contract and you know it won't emit exceptions, following the advice of this Item and declaring it `noexcept` is easy. For functions with narrow contracts, the situation is trickier. For example, suppose you're writing a function `f` taking a `std::string` parameter, and suppose `f`'s natural implementation never yields an exception. That suggests that `f` should be declared `noexcept`. Now suppose that `f` has a precondition: the length of its `std::string` parameter doesn't exceed 32 characters. If `f` were to be called with a `std::string` whose length is greater than 32, behavior would be undefined, because a precondition violation *by definition* results in undefined behavior. `f` is under no obligation to check this precondition, because functions may assume that their preconditions are satisfied. (Callers are responsible for ensuring that such assumptions are valid.) Even with a precondition, then, declaring `f noexcept` seems appropriate:

```
void f(const std::string& s) noexcept;                    // precondition:
                                                          // s.length() <= 32
```

But suppose that `f`'s implementer chooses to check for precondition violations. Checking isn't required, but it's also not forbidden, and checking the precondition could be useful, e.g., during system testing. Debugging an exception that's been thrown is generally easier than trying to track down the cause of undefined behavior. But how should a precondition violation be reported such that a test harness or a client error handler could detect it? A straightforward approach would be to throw a "precondition was violated" exception, but if `f` is declared `noexcept`, that would be impossible; throwing an exception would lead to program termination. For this reason, library designers who distinguish wide from narrow contracts generally reserve `noexcept` for functions with wide contracts.

This is particularly relevant to the C++ Standard Library as library implementers will want the flexibility either to strengthen the `noexcept` guarantee where that makes sense or perhaps instead to provide a wide contract for the implied implementation, possibly including one that doesn't throw in the published contract.[23]

---

[23] Just days before [Lakos22a] was going to print (c. September 2014), Scott Meyers called me at home late one night to ask what all the hubbub was about regarding putting `noexcept` on *narrow* contracts. I said that doing so was a bad idea. He said that everyone was saying that it's not and that the only reason you want to avoid doing it in the Standard is so you can use throwing an exception to test a contract check in your company's implementation of the Standard Library. After mumbling a few words inappropriate to repeat here, I shared with Scott some insights along the lines of this paper's previous section. I then bade him not to use the reason he was about to use, i.e., *facilitation of negative* testing (see "Facilitating *Negative Testing*"). Instead of explaining it the way I suggested, he took another route, which was to

## 4.6 Checked Builds

Until now, we have deliberately avoided using anything related to the Contracts MVP, which is currently under development in SG21 and expected in time for C++26. The point of this paper is to show why the Lakos Rule is important *irrespective* of contract-checking build modes. The only significant difference between a *wide* implementation and checked build modes is that the overhead of checking in a *wide* implementation can be removed in an unchecked build with absolutely no affect on a correct program.

For concreteness, consider again the (slightly simplified) public contract for the bracket operator defined for `std::vector` (ignoring the optional allocator parameter):

```
template <class T>
T& vector::operator[](std::size_t index);
    Return the element at the specified index.  The behavior is undefined
    unless `index < size()`.  Throws nothing.
```

Let's now look at two conforming implementations for the *narrow* contract published for this bracket operator:

```
#include <cassert>                                              // version A 2.0
template <class T>
T& vector::operator[](std::size_t index) {
    assert(index < d_size);                  // build-dependent defensive check
    return d_array_p[index];
}

#include <stdexcept>                                            // version A 3.0
T& vector::operator[](std::size_t index) {
#ifndef NDEBUG                               // build-dependent defensive check
    if (!(index < d_size)) throw std::range_error("[]");
#ifndef NDEBUG
    return d_array_p[index];
}
```

Observe that, because the Standard currently defines the bracket operator not to throw (in contract), either of these implementations is conforming. Had the Standard defined the bracket operator to have a nonthrowing exception specification just because the operator doesn't throw according to its published contract, then both of these implementations would be conforming, but the second one would be tantamount to calling `std::terminate()` on any contract violation.

Understand that it is never acceptable for an exception to escape from a function having a nonthrowing exception specification, especially if such might

---

explain that following the Lakos Rule enabled the implementer with full flexibility to widen the contract in the implementation to be as useful to the client as possible.

be allowed to happen in one successful build but not the other.[24] Were such a thing possible, it would mean that same program built, say, with and without checking enabled could execute radically different code paths having radically different observable behaviors. As a result, the program might work perfectly in a checked build and expose a bug only in an unchecked one. For a dispositive proof of this bold claim, see [P2834R0].

# 5 The Need for Throwing Contract-Checking Violation Handlers

Until now, the reasons given for wanting to follow the Lakos Rule have focused on conventional software engineering principles that are independent of any particular contract-checking facility. Given that SG21 is designing a robust mechanism to be standardized that will vastly improve on the coarse user control offered by its ancient ancestor, the C `assert` macro, the Standards Committee *must* offer all flexibility that naturally and usefully fits the new paradigm. Ignoring or even further postponing such functionality would do a gross disservice to the greater C++ community.

In particular, both business and engineering requirements exist for throwing a user-specified exception when a contract-violation is detected at run time. The current design of the MVP for the Contracts facility in all likelihood will have a link-time-replicable contract-violation handler function that will be handed the current state of a violation in the form of an attribute object of type `contract_violation`. It will then be up to that function to do as it sees fit. The remainder of this section argues for why this replicable violation handler function should have a *throwing* exception specification — i.e., should *not* be specified as `noexcept` in the Standard.[25]

## 5.1 Recovery

Saying that once we have a logic error, the program is defective and we must terminate immediately is easy. But in the real world, that answer doesn't always satisfy the business needs of the organization, let alone the customer. In safety-critical applications, none of this applies. There simply is no such thing as a *fault-tolerant* program, only *fault-tolerant* distributed (sometimes geographically) systems.[26]

---

[24] We are thinking about proposing a language contract check that, throwing through a `noexcept` would invoke the handler, but there would be no ability for a thrown exception to pass that boundary.

[25] A similar argument can and will be made that this violation-handler function prototype should also not be declared in the Standard to have the `[[noreturn]]` attribute: A returning handler is needed to support the *observe* (as compared to the *enforce*) semantic, which has been demonstrated to be useful in a wide variety of non-safety-critical industries (e.g., gaming and desktop publishing) and applications (e.g., power-point plugins (see [Schoedl]) and long-running static-analysis tools, such as Coverity).

[26] [Lakos22a], *S*ection 3.1.“`noexcept` Specifier,” “Potential Pitfalls,” “`noexcept` versus nofail,” pp. 1122–1123

At the other end of the spectrum, we have safety-noncritical systems for which the worst thing that can happen is that the program stops, even if what it is doing isn't quite right. Imagine that you are playing a video game, and you are racking up points. Now imagine we have a contract-checking facility that is on in production, and it detects library undefined behavior. The question that must be asked is, what is the best course of action (1) for the game supplier and, in turn, (2) for the end user?

In the case of the game, maybe the score isn't quite right or some figure isn't rendered properly for a half second. Either way, that's a much better result than terminating the game. The other question is, what's the worst that could happen if *library* UB turned into *language* UB? For programs that are connected to the internet, the risk of language UB is qualitatively higher than those that are not.

In particular, consider an application, such as Coverity, whose purpose is to do extensive static analysis on programs, during runs that can last hours or days. About the worst thing that can happen during a Coverity run is for the program to just stop. Hence, it is designed such that it does its absolute best not to do so.

Consider this real-world anecdote taken (with permission) directly from the SG21 reflector of the C++ Standards Committee:

> A product I previously worked on has an internal conditional assertion facility that, by default, aborts, but where every such assertion can be individually disabled at run-time by setting an environment variable to an appropriate value keyed to the source location of the assert. Programmers using these asserts are expected to provide some kind of reasonable fallback in the event execution continues beyond the failed assert; often this means abandoning some work in progress. That is unfortunate, but far less unfortunate than a customer having to report a high priority production down support case that requires the delivery of a patch to address. The product is not itself a safety critical application, so this kind of graceful degradation is reasonable. Of course, in practice, programmers are not particularly good at ensuring they provide a fallback so employing these workarounds sometimes results in continued execution running straight into UB and a crash report. That is unfortunate of course, but the ability to at least try such workarounds to get a customer unstuck to take the pressure off rushing a (possibly low quality) fix is incredibly valuable.
>
> — Tom Honermann, May 17, 2023

Between these two extremes are industries and applications for which deliberately and abruptly halting — a.k.a. failing fast — may or may not be appropriate. Consider, for example a hedge fund that does algorithmic trading.

Although not safety critical, vast amounts of monetary damage could result if the programs executing the trades are have defects. A classic example of catastrophic failure was the notorious $440MM trading glitch reported by Knight Capital on August 2, 2012.[27] In most other non-safety-critical use cases, however, at least trying to save the current state of the computation — along with any unsaved user data — before gracefully exiting the process might be deemed *safe enough*[28]:

> A contract violation is best handled by a separate system (a different process, or better yet, a separate processor). However, there isn't always a second separate "system" to which we can delegate the handling of the "fatal" error, so we must somehow proceed. The Linux kernel is one such example. I have seen critical financial systems that are not allowed to terminate unconditionally because that might leak objects representing financial entities. Examples that I have heard of but not personally experienced tend to come from relatively small critical embedded systems, such as scuba equipment.
>
> — Bjarne Stroustrup

In his paper, "Unconditional termination is a serious problem,"[29] Stroustrup appeals for something more than the binary alternatives offered by classical facilities, such as the standard C `assert` macro, and what had until recently been proposed as the only two build modes that would be available in the MVP scheduled for C++26[30]:

1. ***No_eval***: compiler checks the validity of expressions in contract annotations, but the annotations have no effect on the generated binary. Functions appearing in the predicate are odr-used.
2. ***Eval_and_abort***: each contract annotation is checked at runtime. The check evaluates the corresponding predicate; if the result equals false, the program is stopped [and] an error return value.

Clearly this minimal functionality does not provide any middle ground for a program that might choose not to terminate immediately upon a precondition failure.

The use of exceptions to signal contract violations is established practice in a variety of popular languages, such as Ada,[31] Python, Java, JavaScript, Ruby, and PHP, just to name a few.[32] In particular, Ada provides a solid foundation

---

[27] https://archive.nytimes.com/dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/
[28] [P2698R0]
[29] [P2698R0]
[30] [P2521R2]
[31] [P2698R0]
[32] Andrew Tomazos, C++ Standards Committee member, SG21 reflector, May 13, 2023. See also [P2853R0], Section 4.3.

for some application that might choose to model recovery from unlikely, systemic, and uniform errors, such as memory exhaustion or contract violations[33]:

> Ada SPARK may be the most widely used contract system in critical applications; it uses exceptions to report run-time contract violations. Ada2002 has adopted that into the standard in the form or the `Assertion_error` exception.
>
> — Bjarne Stroustrup

In short, the use of the C `assert` macro is entirely inadequate for situations in which an organization has made the decision that *fail fast* is not a viable option for either itself or its clients. Until now, the only alternative was a home-grown solution, typically implemented in terms of macros. The Contracts MVP, assuming it is implemented in terms of a link-time user-replaceable global violation-handler function, provides all the machinery to allow the owner of `main` (who presumably also oversees the build modes) to provide the ability to throw an arbitrary exception when a violation is detected at run time. Further delaying such needed functionality would unnecessarily restrict what could be a much more widely used feature.

## 5.2 Negative Testing

The most obvious, concrete, and easy-to-explain motivation for insisting the Lakos Rule be followed, especially in the C++ Standard Library, is that it provides an effective way for implementers of a specification to widen the implied contracts of their implementations in any way they deem appropriate to best support user-friendly behavior when a library function is called *out of contract*. One such friendly behavior would be to throw an exception, e.g., `std::logic_error` or `std::range_error`, appropriate to the user misuse of the library function.

For example, suppose we have a function, `sqrt`, and we want to provide a wide implementation that throws `std::range_error` if the function is called with a negative number:

```
#include <cmath>      // `std::sqrt`                              // Version A1.0
#include <stdexcept>  // `std::range_error`

double mySqrt(double value)
    // Return the positive square root of the specified value rounded up.
    // The behavior is undefined unless `value` is nonnegative.
{
    if (value < 0) throw std::range_error("Hey, you! I said nonnegative!");
    return std::sqrt(value);  // narrow square-root routine in `<cmath>`
}
```

[33] [P2698R0]

The wide implementation unconditionally checks the precondition and, if it is violated, always throws a specific exception with a specific message.

As with any reliable library software, if the function isn't thoroughly unit tested, it doesn't exist. The question becomes, what would be the easiest, most effective way to test that this implementation works the way we intended?

There are some pathologically difficult ways to test wide implementations of `noexcept` functions having a narrow public contract that would have made Rube Goldberg proud, but for this non-`noexcept` function, by far the easiest, most straightforward, practical, cost-effective, and thorough way to test the implementation would be to call the function both in and out of contract and verify its essential behavior directly:

```
#include <cassert>  // standard C `assert` macro

int main()  // Example of a minimal ("breathing") unit test.
{
    // ...                     // Initially, we invoke `mySqrt` *in* *contract.
    assert(4 == mySqrt(16));
    assert(3 == mySqrt( 9));
    assert(2 == mySqrt( 4));
    assert(1 == mySqrt( 1));
    assert(0 == mySqrt( 0));  // Tests below this line are *negative tests*.
    try {       mySqrt(-1); assert(0); } catch (std::range_error&) { }
    try {       mySqrt(-2); assert(0); } catch (std::range_error&) { }
    // ...
    try {  mySqrt(-1e-100); assert(0); } catch (std::range_error&) { }
    try {  mySqrt(-1e+100); assert(0); } catch (std::range_error&) { }
    // ...
}
```

Now suppose that we are using the proposed Contracts MVP anticipated for C++26. To better serve our clients, we've decided to replace the hard-coded wide implementation above with a build-dependent one, using the new Contracts MVP. Note that we can do this because the published contract is

narrow and doesn't say anything about what happens when a client mistakenly calls the library function out of contract[34]:

```
#include <cmath>  // `std::sqrt`                                  // version A2.0

double mySqrt(double value)
    // Return the positive square root of the specified value rounded up.
    // The behavior is undefined unless `value` is nonnegative.
{
    [[ assert: 0 <= value ]]  // build-dependent defensive check (might throw)
    return std::sqrt(value);  // narrow square-root routine in `<cmath>`
}
```

The Contracts MVP will ideally allow the application owner — i.e., the owner of `main` and typically in charge of the build — to install some form of contract violation handler:

```
void std_violation_handler(/*...*/);

void assert_attr(bool b) { if (!b) std_violation_handler(/*...*/); };
    // This function maps to the `[[ assert ]]` contract-checking annotation.
```

---

[34] As of this writing, the syntax of the MVP has not yet been decided, but for most discussions, we have been using the same attribute-like syntax adopted for C++20 contracts, which were subsequently removed prior to its release. Irrespective of the syntax, there are three distinct types of contract-checking annotations (CCA): `pre`, `post`, and `assert`. The first two are intended as decorations on the first declaration of the function.

```
double sqrt(double x) [[ pre: 0 <= x ]] [[ post r: 0 <= r ]];
```

The third is intended for use anywhere within in the body of the function and can be used to *insulate* clients from having to recompile when the implementation of a precondition or postcondition changes:

```
double sqrt(double x)
{
  [[ assert: 0 <= x ]]  // insulated precondition
  int r = std::sqrt(x);
  [[ assert: 0 <= r ]]  // insulated postcondition
  return r;
}
```

Other uses for the `assert` variant exist, such as nesting a partial check within an algorithm (e.g., binary search) so as to avoid having to check the precondition (sorted range) thoroughly or all at once up front.

For the purpose of this demonstration, we have chosen to use the `assert` kind of CCA as it most naturally models what we would have done with a home-grown implementation that throws, but we could have just as easily written the contract check using the `pre` kind of CCA:

```
double mySqrt(double value)
        [[ pre: 0 <= value ]]  // build-dependent defensive check (might throw)
    // Return the positive square root of the specified value rounded up.
    // The behavior is undefined unless `value` is nonnegative.
{
    return std::sqrt(value);  // narrow square-root routine in `<cmath>`
}
```

Now if, in a checked build, the contract-checking annotation (CCA) above — `[[ assert: 0 <= value ]]` — is ever observed to be violated, the user-supplied (or else default) contract-violation handler will be invoked.

Given this framework, how should we go about thoroughly testing our implementation? The best way by far is essentially the same as above. The only difference is that we first need to configure the violation handler so that it, if a contract violation occurs, calls our implementation of the `std_violation_handler` to throw our `MyTestException`:

```
#include <cassert>    // standard C `assert` macro

struct MyTestException { /*...*/ };

void std_violation_handler(/*...*/) { throw MyTestException(/*...*/); }

int main()
{
    // ...
    assert(4 == mySqrt(16.0));
    assert(3 == mySqrt( 9));
    assert(2 == mySqrt( 4));
    assert(1 == mySqrt( 1));
    assert(0 == mySqrt( 0));
    try {       mySqrt(-1); assert(0); } catch (MyTestException&) { }
    try {       mySqrt(-2); assert(0); } catch (MyTestException&) { }
    // ...
    try {  mySqrt(-1e-100); assert(0); } catch (MyTestException&) { }
    try {  mySqrt(-1e+100); assert(0); } catch (MyTestException&) { }
    // ...
    assert("I made it to the end");
    //assert("I made it to the end" && 0);
}
```

Note that, for this test to work as designed, we need to make sure that we are in a checked build mode (either *observe* or enforce) lest we will violate our implementation's (narrow) contract and be subject to full-on *language* UB.

Now consider what would have happened had we decided that, since `sqrt` is never going to throw in contracts, we can just make this *narrow* contract `noexcept`. First, we would no longer be able to supply the helpful value-added, improved QoI that throwing an exception offers. Instead, we would slam against the `noexcept` barrier, and the program would be forced to terminate.

But more to the point of this section, we've just made it a whole lot harder to test the remaining functionality in our wide implementation (which has now been basically relegated to a C-style `assert`). In case you are thinking that you can just comment it out for testing, that doesn't work. Again, for the same reasons stated above, we simply cannot use conditional compilation to elide the nonthrowing exception specification in a checked build, even if just for testing

purposes, because we would be testing some other functionality, not the functionality under test.[35] Practical experience also confirms this claim[36]:

> Having thought more and having grown wiser, `_NOEXCEPT_DEBUG` was a horrible decision. It was viral, it didn't cover all the cases it needed to, and it was observable to the user — at worst changing the behavior of their program.
>
> — Eric Fiselier, on the libc++ switching to death testing

Having a throwing handler is not the only way to test a wide implementation that employs the proposed new contract-checking facility's MVP, but it is by far the most straightforward, efficient, and portable one. Moreover, as previously suggested, having a throwing handler is established practice in a variety of well-known languages (see Section 5.1). Violating the Lakos Rule, on the other hand, would render this testing strategy moot, and we would have to adopt something else. Spoiler alert: The landscape for such alternatives is bleak.[37]

## 6 POTENTIAL PITFALLS OF USING NOEXCEPT

The `noexcept` specifier is among the most notoriously unsafe features of C++. Hence, it unsurprisingly resides in Chapter 3, "Unsafe Features," of *Embracing Modern C++ Safely*.[38] There are many (too many) reasons why the `noexcept` specifier is easy to misuse and hard to use (or know when to use) safely and profitably.

In *Embracing Modern C++ Safely*, a potential pitfall is characterized as a latent design or coding defect that can compile, link, run, and potentially even pass unit testing and peer review, yet likely require subsequent remedial rework if and when it is discovered. The known pitfalls[39] for the `noexcept` *specifier* feature (as distinct from the `noxcept` *operator* feature in Chapter 2) include overly strong contracts guarantees, conflating `noexcept` with nofail, accidental terminate, forgetting to use the `noexcept` operator in the `noexcept` specifier, imprecise expressions in a `noexcept` specification, unrealizable runtime performance benefits, and theoretical opportunities for performance improvement.

---

[35] [P2834R0]
[36] [Khlebnikov2023]
[37] For the best, most comprehensive compendium of knowledge delineating why throwing from a contact-violation handler is by far the best, most cost-effective, and portable means of validating a defensive precondition check, see [P2831R0].
[38] A robust treatment of all of the uses and misuses of the `noexcept` specifier are covered in [Lakos22a], Section 3.1."noexcept Specifier," pp. 1085–1152. Please refer to that reference as a reliable source of truth.
[39] [Lakos22a], Section 3.1."noexcept Specifier," "Potential Pitfalls," pp. 1112–1143

In addition to pitfalls, EMC++S makes note of annoyances,[40] i.e., issues with the feature that do not rise to the level of pitfalls but nonetheless indicate suboptimalities associate with its use. The known annoyances stemming from attempted use of the `noexcept` include algorithmic optimization being conflated with reducing object-code size, code duplication, exception specifications not being part of a function's type, ABI changes in future versions of C++, and exception specifications not triggering SFINAE.

## 6.1 Overly Strong Contract Guarantees

Recall that contracts are binding agreements between each user of a function and its provider.[41] Once we promise something as part of its essential behavior, we are obliged to continue providing it in perpetuity or else default on our implicit obligation to provide stability (in the form backward compatibility) with our existing client base.

When it comes to promising that a function will never throw, the `noexcept` provides the strongest possible guarantee in three ways.

1. It states explicitly and unequivocally that this function does not now and never will throw an exception.
2. It makes that information programmatically accessible to an unbounded audience.
3. It enforces the behavior in the language such that an exception simply cannot — even for a minute during debugging in development —escape from the function.

Placing `noexcept` on any function, even one with a *wide* contract, might well have unintended implications with respect to its backward-compatible extensibility. In fact, before we would even think of declaring a function `noexcept`, we would first have to be 100 percent comfortable stating directly in the contract that this function, "does not throw."

For example, a function whose *essential behavior* requires it to return on every value will be expected to do so:

```
int half(int value);
    // Return an integer that is numerically half of the specified value
    // rounded toward zero --- i.e., half(-3) is -1, not -2.
```

The contract makes clear that there is no flexibility, no implementation-defined behavior, no need to allocate resources, and thus no need to throw. Even without saying the words, "doesn't throw," or "throws nothing," we can be sure that this function is intended to be *nofail*. By adding a specific statement that

[40] [Lakos22a], Section 3.1."noexcept Specifier," "Annoyances," pp. 1143–1150
[41] [Lakos22a], Section 3.1."noexcept Specifier," "Potential Pitfalls," "Overly strong contracts guarantees," pp. 1112–1116

this function does not throw, we give undue weight to that property. For functions that are at a higher level, we might not *plan* to throw any functions, but perhaps someday we might find that some subfunctionality we add later turns out to allocate memory. At that point, we're no longer in a position to guarantee that the function will *never* throw, even if it is extremely unlikely and unimportant.

Consider the function, `businessSort` that sorts its record elements in place according to some subset of its fields:

```
void businessSort(Record* start, int n);
    // Modify the range of contiguous double values beginning at the specified
    // start address and extending n elements so that it is sorted in
    // ascending order in time that is O(n * log(n)).
```

Imagine that, at some later point, the business requires that this sort become stable. No known implementation satisfies the performance requirements with allocating additional memory. Hence, this sort originally could not fail, but now it can. Had we explicitly stated, "doesn't throw," we'd have overpromised. For cases in which memory exhaustion is not an issue, if instead we just don't mention that it throws something, we're implicitly saying that we don't intend to throw, but if we happen to hit memory exhaustion, throwing `std::bad_alloc` is a remote possibility.

Let's now consider again the `sqrt` function:

```
double sqrt(double x);
    // Return the positive square root of the specified `x`.
    // The behavior is undefined unless `x` is nonnegative.
```

This function has a *narrow contract*. If we were to explicitly say "does not throw," we would not be saying the same thing as `noexcept` because "does not throw" applies only to the range that has defined behavior. Still, saying something we don't mean is not a good idea. Consider that the contract, as stated, says that the only precondition is that the input `x` be nonnegative. Well, is a NaN considered non-negative? One could make an argument that it is. (Had we written the precondition as, "the behavior is undefined unless `0 <= x`," then a NaN could much more easily be seen to be out of contract.) Now, if we decide to add to the essential behavior of this contract that the function throws if it is passed a NaN, how can we reconcile that if we already said it doesn't throw in contract?

Again, before we consider adding an explicit statement, "does not throw," to any contract, we need to convince ourselves that we really truly are not ever going to want to extend this contract to one that might, even vanishingly rarely, throw. Once we've done that, then if the contract is wide, adding `noexcept` would not affect the letter of the contract, but it would affect the contract's enforcement in two ways.

1. The essential behavior is now programmatically accessible to an unbounded number of potential clients. This programmatic dependency further cements into the contract that the contract can never, ever throw.
2. Because the language itself interdicts exceptions from passing a nonthrowing exception specification boundary, our inability to throw an exception — even if we wanted to — is mechanically enforced.

In short, most of the time not stating that something throws is good enough to know that it doesn't throw or is unlikely to throw, other than `std::bad_alloc`. In cases in which not throwing is essential for the client to know, stating that the function doesn't throw (in contract) is more than sufficient. The use of `noexcept`, therefore, is needed only when we have reason to believe that generic client code will be using the `noexcept` *operator*, directly or indirectly, to instantiate a better algorithm if it can be known, at compile time, that invoking a particular function (on a particular set of arguments) is not going to throw.

## 6.2 Accidental Terminate

An exception thrown from a `noexcept` function will cause `std::terminate` to be invoked. Two possible scenarios lead to accidental termination.[42] Both involve using code from a third-party library. (Our code examples illustrate Scenario 2.)

1. You write a `noexcept` function that uses a third-party function whose contract explicitly says it doesn't throw.
2. You write a `noexcept` function that uses a third-party function whose contract *says nothing about throwing exceptions.*

### *6.2.1 Scenario 1*

You use a third-party `noexcept` function in your own `noexcept` function, and the third-party contract explicitly says it doesn't throw. Errors result and return error codes. Under circumstances not tested by the third party, your function could throw an exception. Because you made your function `noexcept`, your program terminates unexpectedly, and this termination eliminates any opportunity for you to handle that exception.

### *6.2.2 Scenario 2*

You use a third-party `noexcept` function in your own `noexcept` function, and the third-party contract says nothing about throwing exceptions. The current implementation doesn't throw, so you assume it never will and use it in your `noexcept` function. The third-party vendor might eventually discover a bug and decide to add a throw. Again, because you decorated your function with the `noexcept` specifier, this change causes your program to terminate unexpectedly, which again means that you can't handle the exception:

---

[42] [Lakos22a], Section 3.1."noexcept Specifier," "Potential Pitfalls," "Accidental `terminate`," pp. 1124–1128

```
// From <thirdparty.h>:
double g(double a, double b);
    // Do a calculation and return -1 on error.

// Code under development:
double f(double a, double b) noexcept
    // Do a slightly different calculation and return -1 on error.
{
    double c = a + b;
    double d = a - b;
    return g(c, d); // Note: Pass-through status might prove brittle.
}
```

Use of `g` in the example code above creates a dependency on the other library, which contains `g`'s implementation:

```
// In thirdparty.cpp:
#include <iostream> // std::cerr
double g(double a, double b)
{
    // ... (some non-throwing calculation)
    if (error)
    {
        std::cerr << "Some problem occurred\n";
        return -1.0;
    }
    return result;
}
```

Next, let's suppose that, unbeknownst to us, the maintainers of that library take it upon themselves to add additional, file-based logging using a third-party logging library, FILE_LOGGER, that emits an exception when it fails to write to the log file (due to, e.g., issues with permissioning or disk space):

```
// Library code file
g(double a, double b)
{
    // ... (some non-throwing calculation)
    if (error)
    {
        FILE_LOGGER << "Some problem occurred" << FILE_LOGGER_ENDL;
        return -1.0;
    }
    return result;
}
```

If logging ever fails to write to a file and emits an exception, the `noexcept` specification on our `f` will force the entire program to terminate.

One straightforward way to prevent these unexpected terminations is to assume that any function that is not guaranteed *not* to throw might throw and thus wrap every such function in a `try` block:

```
// Code under development:
double f(double a, double b) noexcept
    // Do a slightly different calculation and return -1 on error.
```

```
{
    double c = a + b;
    double d = a - b;
    try
    {
        return g(c, d); // Note: Pass-through status might prove brittle.
    }
    catch (...) { return -1.0; }
}
```

- The more uses of the `noexcept` specifier there are in a codebase, the greater the chances of `std::terminate`. Generic code, in particular, will fall victim to this.
- If a generic library is designed not to use exceptions, that need not be a problem in and of itself.
- If a generic library uses `noexcept` specifiers, client code is effectively forced into a programming style where it must avoid exceptions.

```
#include <cstdlib> // std::abort
#include <exception> // std::terminate_handler, std::set_terminate
static void emergencySave()
    // Make a best effort to save all existing instances of client data to
    // special recovery files. This function is intended to be called by
    // std::terminate during an emergency program shutdown, e.g., if an
    // unexpected exception occurs. Importantly, the save algorithm is
    // designed to work with just 5MB of available memory.
{
    // ... (Save as much client data as possible.)
    std::abort(); // Kill the program immediately.
}
int main()
{
    std::terminate_handler prevTermHandler = std::set_terminate(&emergencySave);

    // ... (application code)

    std::set_terminate(prevTermHandler); // Restore previous terminate handler.
}
```

In the code above, an unexpected call to `std::terminate` anywhere in the application calls the installed `terminate_handler`, namely, `emergencySave`.

```
#include <exception> // std::terminate_handler, std::set_terminate
#include <new> // std::new_handler, std::set_new_handler

static void emergencySave() {/*...*/ } // same as before

static void* reservedMemoryBlock = nullptr; // memory reserved for emergencies

static void handleOutOfMemory()
// Free reserved memory block and call std::terminate.
{
    ::operator delete(reservedMemoryBlock); // Make memory available.
    reservedMemoryBlock = nullptr;
    std::terminate(); // (hopefully) graceful termination
}
```

```
int main()
{
    std::terminate_handler prevTermHandler = std::set_terminate(&emergencySave);

    // Reserve 10MB memory to use during graceful termination.
    reservedMemoryBlock = ::operator new(10U * 1024U * 1024U);
    std::new_handler prevNewHandler = std::set_new_handler(&handleOutOfMemory);

    // ... (application code --- might exhaust memory)

    std::set_new_handler(prevNewHandler); // Restore previous new handler.
    ::operator delete(reservedMemoryBlock);// Free reserved memory.
    reservedMemoryBlock = nullptr;
    std::set_terminate(prevTermHandler); // Restore previous terminate handler.
}
```

The skeletal solution sketched out above, in addition to setting the terminate handler, allocates 10MB at the start of main and registers `handleOutOfMemory` as the new handler. If an allocation fails, `handleOutOfMemory` frees that 10MB to give the `emergencySave` function more than enough headroom to allocate the memory it requires.

# 7 INCREASINGLY DUBIOUS OPTIONAL USE OF THE NOEXCEPT SPECIFIER

The `noexcept` specifier was invented in a hurry (c. March 2010) as a *patch* to enable efficient use of *move* operations in the presence of the strong exception-safety guarantee already promised by the contracts for certain append functions along with the *strong exception-safety guarantee* in the C++03 Standard Library. Since that time, other uses have evolved, some more useful than others. In this section, we have curated several known uses of the `noexcept` specifier in decreasing comparative utility relative to forgoing their use entirely — even on wide contracts — so as to minimize unintended consequences such as *overly strong contract guarantees* and *accidental terminate*.

## 7.1 Declaring Nonthrowing Move Operations

The raison d'être for inventing `noexcept` was to make it easy for a developer to specify to the compiler that the contract for that *copy* function (operator or contractor) guarantees that, invoked properly (*in contract*), the function will never throw an exception.[43]

As a simplified example, consider a vector-like container, `MyVector`, that has a `push_back`-like function, `pushBack`, that also provides the *strong exception-safety guarantee*, meaning that if an exception is thrown while inserting an element, the state of the original object (and ideally the state of the program as a whole) remains unchanged.

[43] [Lakos22a], Section 3.1."noexcept Operator," "Use Cases," "Appending an element to `std::vector`," pp. 635–639

If we check the element at compile time to see whether moving it can throw, and it isn't declared `noexcept`, the only way to be sure when resizing the array is to create a temporary with the new capacity and then (nondestructively) copy all the elements over first in increasing order. If an exception is thrown, RAII will clean up the original object, which was never touched. If that works, we can then destroy all the originals, replace the new block with the old one, and then proceed with enough space to insert the new element. This is what C++03 had to do:

```
template<class T>
void MyVector<T>::pushBack(const T& value)
{
    const std::size_t nextCapacity = d_capacity ? d_capacity * 2 : 1; // no tr.
    // throwing move

    MyVector<T> tmp;                            // may throw
    tmp.reserve(nextCapacity);                  // may throw
    void* address = tmp.d_array_p + d_size;     // no throw
    for (std::size_t i = 0; i != d_size; ++i)   // for each existing element
    {
        void* addr = tmp.d_array_p + i;         // no throw
        ::new(addr) T(d_array_p[i]);            // may throw
    }
    ::new(address) T(value);                    // may throw (last one)
    tmp.d_size = d_size + 1;                    // no throw
    tmp.swap(*this);                            // no throw, committed
}
```

Unfortunately, the algorithm above requires an extra *N* copies (instead of moves) every *log N* inserts, which could be arbitrarily expensive and perhaps result in an embarrassing pause.

If, however, we can programmatically determine at compile time that — in this generic context — a (perhaps destructive) copy operation on the user-supplied element will never throw, then we can use a different, faster algorithm: Create a temporary `MyArray` object, `tmp`, and reserve the new capacity. If that works, place the new element at the end. If at any point up until now something throws, RAII will kick in and nothing happens. Otherwise, we're fine because copying the rest of the elements over isn't going to throw[44,45]:

```
template<class T>
void MyVector<T>::pushBack(const T& value)
{
    const std::size_t nextCapacity = d_capacity ? d_capacity * 2 : 1; // no tr.

    if (noexcept(::new((void*)0) T(std::move(*d_array_p)))) // is no throw?
    {                                                       // nonthrowing move
        MyVector<T> tmp;                            // may throw
        tmp.reserve(nextCapacity);                  // may throw
```

[44] [Lakos22a], Section 3.1."noexcept Specifier," "Use Cases," "Declaring nonthowing move operations," pp. 1094–1097
[45] [Lakos22b]

```
        void* address = tmp.d_array_p + d_size;    // no throw
        ::new(address) T(value);                   // may throw (last one)
        for (std::size_t i = 0; i != d_size; ++i)  // for each existing element
        {
            void* addr = tmp.d_array_p + i;          // no throw
            ::new(addr) T(std::move(d_array_p[i]));  // no throw (move)
        }
        tmp.d_size = d_size + 1;                   // no throw
        tmp.swap(*this);                           // no throw, committed
        return;                                    // early return
    }

    // throwing move
    // ...
    // ... classic (C++03) implementation elided
    // ...
}
```

Notice that once we get to the point where we have allocated the new block and constructed the new component, only because of the non-throwing exception specification can we confidently proceed to just move all the old elements over, knowing that no exceptions are forthcoming.

More generally, the only candidate suitable for a `noexcept` specifier to achieve its intended purpose is copy operation, which is generally limited to six possible function types, namely: move constructor, move assignment, copy constructor, copy assignment, member swap, and (at least for now) global swap. If your reason to add `noexcept` is not to improve algorithm performance using the `noexcept` operator in a generic context, then you don't *need* to use the `noexcept` specifier.

## 7.2 A Wrapper that Provides `noexcept` Move Operations

The real world is not always like what is taught in school. (More accurate, it rarely is.) As pragmatic software engineers, we sometimes have to choose between the lesser of two evils. Suppose you're running a batch, back-office operation. There is nothing safety critical or even monetarily risky. All you need to do is get the job done with maximum throughput.

You have a long-running application that makes use of modern C++ containers, but you find yourself using a variety of older, legacy components, some of which allocate resources on copy construction and some don't, but almost none of them, when recompiled under C++11 or higher, default to having nonthrowing *move* semantics. Many of the folks who wrote these components have moved on, and they weren't especially careful at documenting or testing their code. (Perhaps that's part of the reason they've gone.)

You have the some of the largest machines money can buy, and exhausting all memory is virtually unimaginable. Moreover, if you did run out of memory, you would want to know that darn quickly, so you could go buy an even bigger

machine, rather than silently plodding on. Besides, the software crashes regularly, so you have to restart it often anyway. What should you do?

One solution might be to essentially lie to the Standard Library and claim that none of your C++03 vintage components ever throw. It's almost true, and you've made the informed decision that if one does throw, you're fine with having your program abruptly terminated without warning. Here is what such a wrapper might look like[46]:

```
#include <utility> // std::move, std::forward

template <typename T>
class NoexceptMoveWrapper : public T
{
  public:
    NoexceptMoveWrapper() = default;
    NoexceptMoveWrapper(const NoexceptMoveWrapper&) = default;
    NoexceptMoveWrapper& operator=(const NoexceptMoveWrapper&) = default;
        // defaulted implementations

    NoexceptMoveWrapper(const T& val) : T(val) { }
        // implicit copy from a T

    template <typename... Us>
    explicit NoexceptMoveWrapper(Us&&... vals)
    : T(std::forward<Us>(vals)...) { }
        // perfect forwarding value constructor

    NoexceptMoveWrapper(T&& val) noexcept : T(std::move(val)) { }
    NoexceptMoveWrapper(NoexceptMoveWrapper&&) noexcept = default;
    NoexceptMoveWrapper& operator=(NoexceptMoveWrapper&&) noexcept = default;
        // override the default exception specification to be nonthrowing
        // moves terminate if the corresponding T operation should ever throw
};
```

As the last three lines of the example wrapper above illustrates, we use the C++23 Standard to *override* whatever the compiler thinks the default exception specification should be to be unconditionally `noexcept`. If a *move* operation ever throws, the program will be forced to terminate. Sometimes an engineer has to make choices, and this use of the `noexcept` specifier is a perfectly reasonable and arguably *necessary* one. Importantly, it's *not* a violation of the Lakos Rule.

## 7.3 Callback Frameworks

A relatively unusual situation where the `noexcept` specifier can potentially make the client's life simpler is a framework that is configured by supplying callbacks. A recent example of this sort of use case came to light when Dietmar Kühl pointed out that the way senders and receivers are implemented in C++ fits this niche use case. Before digging too far into the details, the way the

[46] [Lakos22a], Section 3.1."noexcept Specifier," "Use Cases," "A wrapper that provides `noexcept` move operations," pp. 1099–1101

sender/receiver paradigm breaks things down, as I understand it, you have the ability to supply a principle callback function, `f`, and two additional callbacks, `g` and `h`. The `g` callback is supplied to handle the output, and the `h` callback is invoked if an exception is thrown.

Consider implementing the "then" sender, which per Dietmar, "essentially sets up an object by moving/copying around objects. If anything throws during that time, nothing interesting happens. The real business is, essentially, its actual operation which consists of calling a function, `f`":

```
void someThread(/*...*/)
{
    // ...
    state s = connect(just() | then(f), some_receiver());

    // Here we have an operation state s which can be started:

    start(s);
}
```

Roughly, the general implementation is something like this (forwarding of `args...` omitted):

```
void set_value(state&& self, auto&&... args)
{
    try {
         set_value(std::move(self.receiver), self.f(args...));
     }
     catch (...) {
         set_error(std::move(self.receiver), std::current_exception());
    }
}
```

The only thing that can throw is `f`. If `f` is known not to throw, the try/catch block is superfluous and can be omitted (and the call to `set_error` never happens). Dietmar points out that, for custom receivers that don't throw, not allowing them to be `noexcept` would mean we must manually write an additional, never-used completion function, `h`.

To abstract the problem to a pattern that is more easily recognizable, I asked Pablo Halpern to write the simplest analogous code that he could think of to capture this use case. His solution was to express the situation as a pair of overloaded function templates such that when `doThisThenThat` is supplied a nonthrowing `f`, no `h` is required:

```
template <invocable F, invocable G, invocable H>
void doThisThenThat(F doThis, G thenThat, H doOnException);

template <invocable F, invocable G>
    requires noexcept(declval<F1>()())
void doThisThenThat(F1 doThis, F2 thenThat) noexcept(noexcept(doThis()));
```

Put another way, when `doThisThenThatto` is invoked with three callback functions, it works for all manner of `f`, but when it is supplied only two callbacks, the code will not compile *unless* `f` is declared `noexcept`.

Note that the second overload does not have a `doOnException` argument and participates in overload resolution only if `doThis` can be programmatically known at compile time — via the `noexcept` operator never to throw, which is made conspicuous by embedding it in a conditional `noexcept` specification for the second overload. If `F2::operator()` is known not to throw but is not marked `noexcept`, then we cannot use the second overload, and we are forced to create a dummy argument for `doOnException`, a likely candidate for which would be the address of `std::terminate`.

We have several ways to work around this problem when we are handed a function with a nonthrowing exception specification that we, as programmers, know will not throw. Perhaps the easiest is just to wrap it in a nonthrowing wrapper function:

```
template <typename F, typename... Us>
explicit void noexceptWrapperFunction(F f, Us&&... vals) noexcept
{
    f(std::forward<Us>(vals)...);
        // perfect forwarding to argument function `f` of void return type `F`
}
```

Finally, if the function is under our own control and is wide, there is no issue with the Lakos Rule. If it is narrow, then we need to ask ourselves whether all the other downsides of violating the Lakos Rule add up to a good reason here, which is unlikely given all the obvious workarounds. For the Standard, my preference would be to design the framework in such a manner that the supplier can omit the third argument, have the presumption that it doesn't throw, and terminate if it does throw. Apart from modestly smaller object code size (which is generally the case), it's unclear what `noexcept`'s benefit is here, and we know the downsides.

### 7.4 Enforced Explicit Documentation

We might sometimes have no expectation that anyone will ever apply the `noexcept` operator to the invocation of a function, and still we feel compelled to tag it with `noexcept` because (1) it's wide, (2) we are sure, beyond a reasonable doubt, that it will never need to throw, and (3) it *must* not throw, so much so that we want to make that statement directly in the code.

We know that saying "does not throw" for a narrow contract means "does not throw in contract" and the published contract stays narrow, whereas placing `noexcept` on a function that otherwise had a narrow public contract would force it to become permanently wide. So, for a wide contract, what is the

difference between saying “does not throw” and just putting `noexcept` on the function?

1. Once you put `noexcept` on the function, that's it. Not only are you stating that this thing will never throw, you can never take it off because now you've made it part of the programmatically accessible interface. Changing it back would be a breaking change, and code might start to behave differently. That's not to say that taking “does not throw” off a contract doesn't change it; it certainly does, which is why, unless you are extremely sure that you *really* mean it, avoiding even putting that in English is wise, because as code, especially application code, evolves, functions that depend on other functions that previously didn't allocate memory might start to and have to break their promise, and that discrepancy will come back to haunt you, even if you didn't agree to it.
2. By placing `noexcept` on a wide contract, you are getting a bit more support than you would with just an English contract. If your library function is called out of contract and it doesn't slam into *language* UB, you can count on this function not throwing past the `noexcept` barrier, or if it does, it’s guaranteed to terminate. Sadly, however, if we do encounter language UB, the function is not required to do anything at all, and, at least in theory, it could even allow a thrown exception to bypass the `noexcept` barrier (but in practice that's probably quite unlikely).

The bottom line is that, if a function has a wide contract that is unlikely to change and if you are certain that it cannot or must not ever throw, then state that explicitly in the contract. If you then feel compelled to decorate it with `noexcept`, be my guest. A typical example in the Standard would be a `const` member function with a wide contract, such as `std::vector::size()`. As ever, for a function having a *narrow contract*, forget about it.

## 7.5 Reducing Object-Code Size

Unfortunately, people are seemingly rewarded when they place `noexcept` on a function and both the object and binary code size goes down. This reduction typically achieves no useful purpose. Making code smaller by removing exception code on the cold path doesn't measurably, let alone significantly, make the code run any faster on average. The only place where binary size matters is embedded systems, and for those, one would typically disable exceptions entirely, rendering any such use of `noexcept` moot.[47]

Just to make a noticeable difference in binary code size, even though doing so accomplishes no useful goal, one would have to use (overuse) `noexcept` widely, thereby needlessly increasing the likelihood of creating overly strong contracts

[47] [Lakos22a], Section 3.1.“noexcept Specifier,” “Use Cases,” “Reducing object-code size,” pp. 1101–1111

and especially by increasing the probability of accidental termination. This reason to use `noexcept` is the least compelling (except, of course, for the final one, below) and should be used only when (1) the function's contract is wide, (2) not throwing is already part of its guaranteed *essential behavior*, and ideally (3) you reasonably believe that code you depend on is not likely to change.

### 7.6 Unrealizable Runtime Performance Benefits

The claim that `noexcept` improves runtime performance — even a little bit — is unfounded and specious. No evidence (theory or data) supports this claim, and copious amounts of both to suggest otherwise.[48,49,50]

All modern platforms use the zero-cost exception model.[51],[52] In this model, literally all the extra cost lies on the cold path in the sense that, if an exception is triggered, the flow of control will deviate from the expected (hot, branch-predicted) path to other, distant code that is potentially not even be paged in. The tradeoff is that not throwing costs you literally nothing in run time, but heaven help you if you throw one.

So, though difficult to believe, anyone who claims that `noexcept` speeds up your code — even someone whose knowledge and/or veracity you (used to) respect — is mistaken. Please never go littering your code with `noexcept` in the futile hope that doing so will speed up your code at all: It will not. In fact, the bell curve of noise due to cache line placement — in either direction — will overwhelm even any pathological performance gains you think you might be realizing.

## 8 The C++ Standard Supports the Multiverse

As software developers, we have each amassed our own, sometimes formidable, design experience. The totality of that experience, however, will be unique to each individual. Developers in a particular industry, such as finance, safety-critical systems, desktop publishing, or gaming, may find synergies and commonalities that simply don't exist across industry boundaries. As a result, what may seem like an obvious "no-brainer" design choice to a competent developer in one industry might be considered an untenable (if not unfathomable) choice by an equally competent application developer in another.

---

[48] [Lakos22a], Section 3.1."noexcept Specifier," "Potential Pitfalls," "Unrealizable runtime performance benefits," pp. 1134–1143
[49] [Dekker19a]
[50] [Dekker19b]
[51] [Lakos22a], Section 3.1."noexcept Specifier," "Potential Pitfalls," "Unrealizable runtime performance benefits," "The zero-cost exception model," p. 1136
[52] [Mortoray13]

After years of experience working with the best developers on the C++ Standards Committee, one comes to learn that two dissimilar design approaches can each be optimal for their respective universes. As Standards Committee members, therefore, our responsibility is to design the C++ language and its Standard Library such that they enable if not support all industries.

Stroustrup refers to the superposition of the respective realities that arise in all relevant industries as the *multiverse*. Hence, rather than inadvertently attempting to maximally satisfy the needs of any one particular industry (e.g., and isn't that all too often our own?), as members of the Standards Committee we must deliberately take a step back and ensure that we are striving to maximize the satisfaction of the *union* of the requirements of all such industries.

The diverse and sometimes conflicting needs of the multiverse abound. For example, C++ should be easy for a novice to understand and use yet enable an expert to do pretty much anything remotely reasonable. In this case, Stroustrup's design advice would be to keep simple things simple yet not preclude more sophisticated use, perhaps at some later time.

Some organizations use C++ primarily because of its runtime efficiency, others for its scalability, and still others due to its ability to get close to the hardware. Some industries, such as medicine and aerospace, involve *safety-critical* systems. An undetected defect in such systems might lead to catastrophic results, even loss of life. Hence, the cost of design, development, testing, and deployment in such industries is typically disproportionately high. Other industries, such as desktop publishing, have no safety-critical components. For products in those industries, an inexpensive development strategy that admits the occasional defect in a new release is often preferred.

As previously discussed, in some industries, such as gaming, an application might have an explicit design goal that it never intentionally self terminates. Stopping the game might be deemed as bad or worse than anything else that might reasonably happen, and there's always the chance that the player won't notice the current defect. In finance, however, failing fast or at least not continuing as if nothing had happened is often preferred over allowing a program to just keep going, possibly losing enormous sums of money for the firm.

These distinct universes do not always overlap, and yet they are all contained in the same multiverse. To provide the widest possible applicability and utility to its prospective clients, the C++ Standard Library along with the C++ Language itself must support this multiverse. That is, C++ must enable design without making a value judgment as to whether a particular design choice is acceptable because such subjective decisions simply cannot be reached in our

multiverse in which the same design might be perfect in one context and entirely unacceptably in another. Instead, the Standards Committee must withhold judgment and instead design with flexibility to serve all industries.

For example, suppose we decided that C-style casts are so bad that we want to deprecate them. Should we do that? Of course not. Why? Some developers want to use them, perhaps for good reason. Moreover, these casts are already in use. Hence, if you think they're bad and shouldn't be used, just don't use them. Live and let live; the multiverse goes on.

When it comes to new features, the bar is a bit higher. In that case, we are typically increasing the complexity of the language for all implementers and users so the test is whether the added functionality provides real capability, theoretical value, or just syntactic sugar. On the other hand, sometimes adding a feature adds little complexity compared to the new capabilities it would bring to many users.

For example, choosing to specify that the contract-violation handler cannot return by specifying it as `[[nothrow]]` in the Standard would aggressively enforce one policy over another, thereby disenfranchising users who would otherwise avail themselves of this capability, which comes essentially free in the current MVP. Similarly, specifying in the Standard that the replicable `violation_handler` function in the MVP is necessarily declared `noexcept` would preclude two important, effective, and widely applicable use cases described earlier (see Section 5).

The Standard must support all the universes of all industries, organizations, and well-intentioned, reasonable developers. Sometimes we will have to choose between two policies. For example, when the expression in the contract check throws needs to be defined to do something, we need to choose what we think will ratify the widest set of use cases without overly inconveniencing the typical or novice user. (That decision will likely be to catch the exception in the contact check, treat it as a contract-violation, and pass the deception along with all the rest of the relevant information into the handler where it could even be rethrown if desired.

In the case of whether to allow continuation or throwing, the answer is simple. Let's say that not allowing something is decision X and allowing something is decision Y. If the Standard imposes X, then anyone who would have preferred Y is just out of luck. On the other hand, if instead we standardize Y, then those who prefer X don't have to do Y, and those that prefer Y can still use that option. The obvious conclusion is that we should standardize a contract-checking handler declaration that allows that handler to (1) continue, (2) throw, and (3) whatever else the user decides is appropriate (because it’s C++ code). By the same token, the Lakos Rule, being the more permissive

requirement, should be standardized, with implementers permitted to strengthen the exception guarantees or otherwise provide a better QoI, e.g., a wide implementation employing the new Contracts MVP.

The design goals, rules, and guidelines for a development team in one industry might be radically different from those for a team in another industry, and yet neither team is doing anything wrong. Different engineering tradeoffs will be made for good reasons. The forces that govern these tradeoffs — e.g., safety, security, government regulation, and economics — will play a dominant role in how programs are designed. Understanding that, given different exigent requirements, smart people will naturally write good C++ programs differently is critically important to designing a maximally useful Standard. Hence, when making a policy decision, standardizing a less restrictive one that still admits the other is often the better choice.

## 9 THE LAKOS RULE

The Lakos Rule (note that I didn't name it) is nothing more than a simple observation based on classic principles of software design in terms of contracts. Over the years, much has been read into the rule, but what makes it elegant and useful is its inherent truth and simplicity. In fact, the rule (observation, really) is the title of this paper:

> The Lakos Rule:
> Narrow contracts and `noexcept` are inherently incompatible.

The obvious interpretation of this rule is that, if you have a contract that is naturally narrow and will absolutely never throw when called *in contract* and if you feel that it's important that developers know this, note in the English documentation that the function “does not throw” or “throws nothing.” Otherwise, do nothing. If this public contract should someday widen or if the implied contract of the implementation is deliberately made wide (possibly only in certain build modes) to detect client misuse (aka *defensive programming*), those options will be preserved.

Beyond that, whether to declare a function having a *wide contract* `noexcept` comes down to whether it is possible to predict whether this function might someday evolve or be ported to a different context where the need to throw an exception in contract might emerge.

Coming from the opposite perspective, having more functions declared `noexcept` than justifiably need to be does nothing to help reduce the likelihood of accidental termination, especially when exceptions are used widely to communicate across multiple levels of function calls.

The `noexcept` specifier is rarely necessary, and when it is needed, it’s for niche uses, almost always having to do with efficient copies in the presence of a

stronger-than-standard exception-safety guarantee. Unless a function is anticipated to be queried at compile time from a generic context using the `noexcept` operator (directly or indirectly), any use of the `noexcept` specifier can be safely omitted and with zero loss in observable average runtime performance.

TL;DR: Avoid declaring a function that would otherwise have a *narrow contract* `noexcept` unless there is a compelling engineering or business reason to do so.

### 9.1 Exception(s) to the Lakos Rule

The original Lakos Rule of 2011 claimed three operations that were given special consideration: move construction, move assignment, and swap. With over a decade of experience, no new compelling examples have been found, and swap does not clearly qualifiy as special.

The move constructor and move assignment operator are special as they are used to relocate elements, and useful guarantees can be given if relocation never fails. Never failing (*nofail*) is a stronger constraint than never throwing, but was deemed a reasonable, if imperfect, approximation for most libraries.

Recall that the `noexcept` operator was conceived solely to preserve the *strong exception-safety guarantee* provided by C++03 when a vector grows by insertion at the end. Especially to expand the capacity of the vector and preserve the succeed-or-no-change semantic, we must guarantee that the relocate operation moving elements from one region of memory to another cannot fail. The performance change switching from copy to move to achieve such a relocate is well known, so we choose to guarantee that move operations do not fail (at least not by throwing), rather than preserving the freedom of potentially throwing in a future implementation of the contract.

Hence, we consider *move operations* in particular to be a very special case such that even if a move operation were somehow narrow, we would nonetheless prefer it to be declared `noexcept` anyway since the property of allowing the `noexcept` operator to be used to query its (non-throwing) exception specification is critically important *essential behavior* for its primary use case (i.e., enabling algorithmically better implementations in generic contexts).

In our decade of using the `noexcept` operator, we have not run into the same sort of benefits for the `swap` function that we were expecting. While best practice remains to write a `swap` function that guarantees to not throw (in contract), and with constant complexity, we have not seen the benefit of making that guarantee in the type system; no algorithm that we have encountered has a different optimal code path given the guarantee. (On the other hand, its contract can often give stronger postconditions when the plain

language contract for the `swap` function gives the nonthrowing — or better, *nofail* — guarantee.)

If there were to be an exception to the Lakos Rule, it would have to satisfy four properties:

1. The operation the function provides has an inherently narrow contract.
2. A primary use case would be lost if it had a throwing specification.
3. To disallow throwing in response to a contract violation is acceptable.
4. No better design alternative is available (or foreseeable).

First, by *inherently narrow*, we are talking about the contract for a function, such as a real `sqrt` or the bracket operator for `std::vector`, for which there is no obvious wide interpretation. Second, a generic client will be expected to employ the `noexcept` operator (directly or indirectly) to determine whether a particular invocation of the function cannot throw so that it may instantiate (at compile time) an algorithmically superior implementation. Third, we need to be fine with the idea that, if we call that function out of contract and an exception is thrown in response, the program will be forced to terminate. Forth, we simply cannot think of any other, better practicable way to solve the problem.

### 9.2 Are `swap` Operations Exceptions to the Lakos Rule?

Let us consider whether the standard `swap` function merits an exception to the Lakos Rule. The primary template, i.e., the basis for customization relying on ADL lookup, has a wide contract that has no nonthrowing constraints in the plain language contract. Hence, it does not have an inherently narrow contract, it should always be wide, and it may throw exceptions when called in contract. Thus, `swap` does not qualify for an exemption from the Lakos Rule, and there is no need to consider the remaining items in our list of properties.

For purpose of illustration, however, we would like to know if there is a primary use case that would query the `noexcept` operator that we would lose without a `noexcept` specification? When we drafted the original rule, we believed that, given the importance of no-throw `swap` to plain language contracts, it would be an important operation to support a nonthrowing exception specification where possible, so we applied a *conditional* exception specification when the move constructor and move-assignment operator had a nonthrowing exception specification.

With a decade or more of experience, though, we have not encountered functions or algorithms that benefit from use of the `noexcept` operator on this operation. Instead, we continue to run into guarantees of the plain language contract that do not lean on the `noexcept` operator itself. Hence, if we were to specify `swap` for the Standard today, it would not merit even its conditional exception specification.

For item 3, we cannot possibly presume that all software built on top of the Standard Library can accept disallowing exceptions in response to contract violations. As a foundation for the C++ ecosystem, our specification cannot reach into user requirements in this way. Hence, `std::swap` fails on the third item as well.

Finally, we come to the question of whether no better design alternative is available. The main alternative we lean on these days is customization-point objects (CPO), and while the complexity of such types goes beyond the simple `swap` template, the user experience seems better. So perhaps if failing on all four, then we'd have to repeat this analysis for the CPO!

### 9.3 Are `move` Operations Exceptions to the Lakos rule?

The other two exceptions to the Lakos Rule that were blessed with conditional exception specification are the move constructor and move assignment operator. Let us again apply the four tests and see if we learn anything.

First, neither the *move constructor* nor *move assignment* operator are *inherently* narrow, although there are some *artificially* narrow move-assignment operators in the Standard Library (more on those assignment operators later). Hence, the first test fails, so we do not get an exception to the Lakos Rule. However, let us continue analysis as before.

For item 2, we ask if we would lose a primary use case if we could not query these operations with the `noexcept` operator. The answer here is a strong affirmative, since the primary use case of the `noexcept` operator in the paper that proposed the feature for the language[53] is to optimize relocation of elements in a vector under a variety of operations such as growing the capacity, inserting, and removing elements, and so on. Hence, even if the contract were somehow *narrow,* we would nonetheless make an exception for move operations because that's the only way in which they can be used for their singular intended purpose. As for the final two check boxes, yes we would be OK with a checked build that terminated the program due to an out-of-contract call to a narrow move, and, no, we don't have a better idea.

The remaining concern that we promised to address is the *artificially* narrow contracts in the Standard Library for move assignment on standard containers where allocators do not propagate and do not compare equal. This constraint is artificial since there is no inherent reason such containers could not move their elements by copying; after all, the motivation for *move assignment* was as an optimization to *copy assignment*, so copying when the optimization is not available should be the expected semantic (move-only types exempted). We note

[53] [N3050]

that the Standard still does not mark these operations as unconditionally `noexcept` but places a conditional `noexcept` on those operators instead that means, for example, `pmr` containers do not have nonthrowing move-assignment operators. Hence, although enabling nonthrowing move-assignment for the `pmr` containers might seem desirable, the artificial rather than inherent constraint does not satisfy the first test, and we still do not apply a `noexcept` specification here.[54]

### 9.4 Are There *Any* Exceptions to the Lakos Rule?

Yes! There is one. Implicit in every contract is the precondition that the client does not pass in an argument having an *indeterminate value* that the function intends to read:

```
int f0() {
    double x;                   // indeterminate value

    double y = std::move(x);    // precondition: `x` is initialized

}
```

The built-in scalar type int has a `noexcept` *move constructor* that requires its argument not to be of *indeterminant value*. Hence that move operation hypertechnically has a precondition such that calling the move constructor with that specific input is *undefined behavior*, and hence the *move constructor* (like pretty much any function that takes any argument) has undefined behavior, and therefore a *narrow contract*! But, as we discussed earlier, a narrow move operation has special privileges because it passes all four of our check boxes. Here is a second example in terms of `std::array`[55]:

```
using A = std::array<float, 1>;

static_assert(std::is_nothrow_move_constructible_v<A>);

int f1() {

    A a0;                   // indeterminate valueb

    A a1 = std::move(a0);   // precondition: a0 is initialized
}
```

Again, the implicit nonthrowing move constructor for `std::array` has a move constructor that expects its source object to be initialized. And again, move is a special case satisfying all four of the requirements: (1) this move function is inherently (albeit barely) narrow, (2) a nonthrowing exception specification is essential to its primary purpose, (3) we would much rather have faster

[54] Note that the authors of [Shearer20] and [P0178R0] intend to address this concern by making move assignment in these cases well defined; the authors are aiming to bring a revised paper to Kona later this year.
[55] Courtesy of Neven Liber, SG21 Reflector, May 16, 2023.

algorithms than try to preserve extensibility or worry over a thrown exception in pathological contract violation (that we can't even test for), and (4) we don't have a plan B.

As a practical matter, we do not consider a contract that would otherwise be considered *wide* to become *narrow* due to irrelevant abuse of the language, such as passing in a destroyed or otherwise unusable object, since that would eliminate any utility in distinguishing *narrow* from *wide* contracts. Hence all the above move functions would be considered wide, and passing in an indeterminant or destructed object is simply a bug.[56]

tl;dr: The Lakos Rule is about as close to absolute as any rule with an exception (pun intended) can be.

# 10 Recommended Use of the noexcept Specifier

No recommendation fits all situations for using `noexcept`; use it where it affirmatively adds value, and don't use it where it doesn't (because we can always add it later) or, worse, where it subtracts value.

## 10.1 The C++ Standard Library

If we were starting from scratch, I would advocate using `noexcept` only where we anticipate standard containers will need it (typically only *copy*, *move*, and *swap* functions). I see no compelling reason to force implementers to do more against their will. Given what we have now, the trend seems to be also to declare `const` member functions having wide contracts that will never throw to be `noexcept`, and that seems nonproblematic yet has the ever-so-slight benefit of producing slightly less (but not faster) code on average.

Hence, our recommendation for the Standard Library *specification*, which is a foundation for almost all C++ software, is that the Lakos Rule be observed for *every* library function having a *narrow contract* and that future flexibility be considered carefully before applying the `noexcept` specifier to nonthrowing wide contracts. On the other hand, move operations have demonstrated the importance of communicating their nonthrowing nature via use of the `noexcept` operator, so even conditionally nonthrowing (but invariably *wide*) move contracts merit applying `noexcept`.

## 10.2 Standard-Library Implementations

In anticipation of the Contracts MVP, we want to strongly encourage (if not mandate[57]) Standard Library developers to follow the Lakos Rule so that they

[56] Note that passing an indeterminant value by value is automatically UB whereas passing an indeterminant value by reference is allowed as long as the function does not attempt to read it; assigning to it and taking its address, however, is not (language) UB.
[57] [P2837R0]

will be able to widen their implementation to incorporate the full capability of a throwing contract-violation handler if they so choose.

That said, it is ultimately the responsibility of each C++ library implementor to make the business and engineering tradeoff of how they want to spend the unspecified behavior afforded them by minimal use of `noexcept` in the Standard Library's specification. In particular, an implementor is free to strengthen the exception specifications of any function whose current contract that doesn't require it to throw (not recommended). Alternatively, implementors may preserve the narrow public contract and instead provide a wide implementation that incorporates the new Contracts MVP, on track for C++26.

Although *narrow contracts* and nonthrowing exception specifications are inherently incompatible, they are not entirely so. If an implementor chose to widen the public contract to permanently disallow any thrown exception from a narrow contract and later add checking that attempted to check the original narrow contract, then some of the benefit of the checking would be realized; the only problem is that, if an application were counting on recovering from a contract-violation calling a standard function, and the implementor has cut off throwing as a possibility in an extended implementation, and then the natural (well-defined) consequence will be program termination.

### 10.3 Third-Party Libraries

The Lakos Rule applies to all software, not just the Standard; hence the general advice to ardently avoid (where unnecessary) placing `noexcept` on functions having a narrow contracts pertains. That said, each third-party library vendor must make an informed decision as to the extent to which optional use of `noexcept` on wide contracts serves their business needs and client base.

As a (strong) general recommendation, a `noexcept` specification should not be used unless the following three conditions are met.

1. Functions are expected to be implemented with an optimized code-path by querying the `noexcept` operator on *this* function.
2. The plain language function contract is wide.
3. The function guarantees not to throw; if the function guarantees not to throw under only specific circumstances, those circumstances can be described by a predicate in the exception specification

When used in software written above foundation layers, pragmatic engineering values may relax even these constraints, such as in an environment in which failing an operation (in any build mode) can terminate the program.

### 10.4 End-User Libraries

The Lakos Rule applies generally, even to end-user software, the only substantive difference being that the organization is in complete control of all

its internal clients; hence, backward incompatibility, though still painful, might not be irreparable. Again, each library implementation will make its own informed determination as to tradeoffs between retaining a narrow contract and permanently assigning requirements for all syntactically valid state and input combinations that explicitly preclude ever throwing for any reason.

## 11 CONCLUSION

This paper provides an in-depth elaborate, tutorial-level support for the simple observation that nonthrowing exception specifications are inherently and fundamentally incompatible and inconsistent with *narrow contracts*. That is, a function that otherwise would have a *narrow contract* loses its narrow status as soon as its exception specification becomes nonthrowing because now there is no syntactically valid combination of input and state values for which the function exhibits *undefined behavior* (i.e., behavior having no requirements).

We started by considering the value proposition of designing software having *narrow contracts* rather than necessarily always making them *wide*. Then, after reviewing some basic principle of classical software engineering, including *Design by Contract* (DbC) and the *Liskov Substitution Principle* (LSP), we explored the benefits of narrow contracts as a means of extending software APIs in a backward-compatible manner over a sequence of versions. We observe that, had we initially declared to be `noexcept` a function otherwise having a narrow contract, the benefits of unfettered backward compatibility were lost --- further proof that the `noexcept` function's contract simply cannot be considered narrow in any practical senses.

In particular, we observed that the ability to provide a *wide implementation* to a narrow public contract serves as an opportunity to render increasingly valuable QoI without affecting the behavior of any programs currently written to that narrow (interface) contract. Only then did we introduce SG21's burgeoning Contracts facility and argue that failure to follow the Lakos Rule in the Standard would disenfranchise (1) clients of the Standard who want to make use of such QoI and (2) implementers of the Standard (or parts thereof) who might want to create effective negative tests without having to resort to a complicated, nonportable, grossly inefficient single ("death") test per thread (or worse, per process).

Next we considered the consequences of just generally inappropriate or unnecessary (over)use of the `noexcept` specifier, especially in the C++ Standard Library. Having more nonthrowing exception specifications does nothing to help avoid accidental termination, especially for applications that make heavy use of exceptions to communicate across multiple levels of function calls.

We then took a hard look at various purported uses of the `noexcept` specifier. Importantly, we observed that any need for the `noexcept` specifier, is rare, highly specific, and typically geared to maximizing *algorithmic* runtime performance in the presence of (dubious) strong exception-safety guarantees imposed by a few (e.g., insert) member functions of standard containers. By contrast, SG21's robust runtime contract-checking facility, for which an MVP is on track to be released for C++26, will offer value in terms of correctness, safety, security, and robustness, for any program that makes use of libraries — especially the C++ Standard Library — having functions with narrow contracts.

Considering that excessive forced use of the `noexcept` specifier adds substantial risk whereas widespread (optional) use of the C++ Contracts MVP does just the opposite, the choice of which to favor seems obvious, yet the Standards Committee does not need to make that choice! By simply following the Lakos Rule in the C++ Standard Library specification, we leave open the possibly for implementers to (1) strengthen the exception specifications themselves or (2) provide a wide implementation that supports optional contract checking in the appropriate build modes or (3) both at the same time. (However, due to the inherent incompatibility identified by the Lakos Rule, a violation handler throwing into a `noexcept` specification will necessarily result in a call to `std::terminate`.)

Next, we restated the Lakos Rule and discussed its application. We then clarified the *Lakos Rule* in the context of all we have learned about modern C++ and proceeded to tested its applicability on some previously thought-to-be exceptions. For completeness, we provided its one and only known (hypertechnical) exception, which involves the passing of *indeterminate values* to a *move constructor* (or *move assignment operator*).

Finally, we provided recommendations for codifying the Lakos Rule as design guidance geared toward the C++ Standard Library specification, implementations of the Standard Library, arbitrary third-party library providers, and ultimately end-user libraries. The conclusion is that Lakos Rule has no known practical exceptions, pertains generally, and is especially important for the Standard Library, which must support the multiverse.

## 12 Acknowledgements

I want to thank MIT Institute Professor of EE&CS Barbara Liskov, Ph.D., (Stanford, 1968) for her pioneering work on abstraction and, in particular, her seminal observation, properly referred to as the *Liskov Substitution Principle*, which has become the gold standard of backward-compatible versioning in software. Separately, I would also like to thank everyone who inadvertently motivated me to write down this cornucopia of rationale for what I thought was "obviously" just sound modern software design.

# 13 APPENDIX

## 13.1 How did we get here?

At the March, 2010, Standards Committee meeting in Pittsburgh, David Abrahams brought to the Standards Committee's attention that, if a move constructor is not known at compile time not to throw, there is no way to use it to its full efficiency and still achieve the requisite strong exception guarantees, e.g., appending to a `std::vector`.

In response to this unanticipated emergency, both the `noexcept` specifier and `noexcept` operator were quickly concocted for the niche use of determining, at compile time, whether a *copy*, *move*, or *swap* operation on a type was nonthrowing. No other use was envisioned.

As this new language feature was being added to the Standard, some began to question whether widening use of this nascent feature might be appropriate to include other functions for which there was no fathomable reason why generic code would ever need to query such a property.

Those intimately familiar with contract-checking in practice knew that, when some form of contract-checking was eventually adopted into the Standard, one of the viable (and useful) behaviors resulting from a detected precondition violation would be to somehow throw a (e.g., user-provided) exception.

The *Lakos Rule*, which in essence states that *a function having one or more preconditions shall* not *have a nonthrowing exception specification*, was articulated to provide the needed appropriately conservative, objective guidance since the hastily conceived `noexcept` specifier was about be summarily distributed across the entire Standard Library during that frantic final meeting in Madrid (March, 2011) before C++11 was shipped. After considerable discussion, this important guidance[58] was adopted with strong consensus (>75%).

Over the years, several attempts to introduce contract checking into the C++ Standard have come and gone and, as yet, the Standard has nothing concrete to show for it. We have, however, learned quite a bit about what is needed to support contract checking at scale and are now poised to propose a solution and a foundation for something that will be truly seminal to safety and correctness in the C++ language.

In the meantime, we've had much discussion about certain alleged benefits of employing `noexcept` much more liberally. A widely touted yet unsupported claim states that declaring an arbitrary function `noexcept` can somehow

[58] [N3248]

measurably, let alone significantly, improve its runtime performance on modern, general-purpose architectures. Any such claims are flat-out wrong. Through controlled experiments and empirical measurement, this conjecture has since been repeatedly and thoroughly debunked (see Section "Unrealisable runtime performance benefits").

These and other misguided beliefs and ideas about what `noexcept` can achieve are likely responsible for the gross overuse of the `noxcept` specifier, which already threatens the safe, valid use of C++ exceptions in practical applications. Excessive unnecessary use of the `noexcept` specifier is destined to become even more problematic and contraindicated once the highly anticipated MVP for a general-purpose C++ contract-checking facility becomes available (expected for C++26).

### 13.2 Structurally Inherited Functions and Contracts

*This section is forthcoming in a future release of this paper.*

### 13.3 `const` Member Functions and Contracts

*This section is forthcoming in a future release of this paper.*

### 13.4 Virtually Functions and Contracts

*This section is forthcoming in a future release of this paper.*

## 14 References


[N3050] David Abrahams, Rani Sharoni, and Doug Gregor, "Allowing Move Constructors to Throw," N3050R1, ISO, Geneva, March 12, 2010 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3050.html

[N3248] Alisdair Meredith and John Lakos, "`noexcept` Prevents Library Validation," N3248, ISO, Geneva, February 28, 2011 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3248.pdf

[N3279] Alisdair Meredith and John Lakos, "Conservative use of `noexcept` in the Library," N3279, ISO, Geneva, March 25, 2011 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3279.pdf

[P0178R0] Alisdair Meredith, "Allocators and swap," P1078R0, ISO, Geneva, February 15, 2016 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0178r0.html

[P1656R2] Agustín Bergé, "'Throws: Nothing' should be noexcept," P2834R0, ISO, Geneva, February 11, 2020
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1656r2.html

[P2300R6] Michał Dominiak et al., "`std::execution`," P2300R6, ISO, Geneva, January 19, 2023
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html

[P2521R2] Gašper Ažman , Joshua Berne , Bronek Kozicki, Andrzej Krzemieński , Ryan McDougall, Caleb Sunstrum, "Contract support — Working Paper," P2521R2, ISO, Geneva, March 15, 2022
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2521r2.html

[P2646R0] Parsa Amini, Joshua Berne, and John Lakos, "Explicit Assumption Syntax Can Reduce Run Time," P2646R0, ISO, Geneva, October 15, 2022
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2646r0.pdf

[P2698R0] Bjarne Stroustup, "Unconditional termination is a serious problem," P2698R0, ISO, Geneva, November 18, 2022
https://www.openstd.org/jtc1/sc22/wg21/docs/papers/2022/p2698r0.pdf

[P2821R0] Jarrad J. Waterloo, "`span.at()`," P2821R0, ISO, Geneva, February 20, 2023
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2821r0.html

[P2831R0] Timur Doumler and Ed Catmur, "Functions having a narrow contract should not be `noexcept`," P2831R0, ISO, Geneva, May 15, 2023
https://isocpp.org/files/papers/P2831R0.pdf

[P2834R0] Joshua Berne and John Lakos, "Semantic Stability Across Contract-Checking Build Modes," P2834R0, ISO, Geneva, May 15, 2023
https://isocpp.org/files/papers/P2834R0.pdf

[P2837R0] Alisdair Meredith and Harold Bott, Jr., “Planning to Revisit the Lakos Rule: The Lakos Rule is Foundational for Contracts,” P2837R0, ISO, Geneva, May 10, 2023

[P2853R0] Andrew Tomazos, “Proposal of std::contract_violation,” P2853R0, ISO, Geneva, April 24, 2023
https://isocpp.org/files/papers/P2853R0.pdf

[Cargill92] Tom Cargill, *C++ Programming Style* (Reading, MA: Addison-Wesley, 1992).

[Dekker19a] Niels Dekker, “`noexcept_benchmark`.” published via GitHub, January 18, 2019
https://github.com/NDekker/noexcept_benchmark/blob/main/LICENSE

[Dekker19b] Niels Dekker, “Lightning Talk: `noexcept` considered harmful???” *C++ on Sea*, 2015
https://www.youtube.com/watch?v=dVRLp-Rwg0k

[gehad17] Gehad Alkady, Hassanein H. Amer, and Ramez M. Daoud, “Remotely configurable fault-tolerant FPGA-based pacemaker,” *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, Cairo, Egypt, 2017, pp. 19–24
https://ieeexplore.ieee.org/abstract/document/8275270

[indepth21] *Jakub Šimánek*, “Building A Fault Tolerant Rebreather: Our Path to Simplicity,” *InDepth*, February 2, 2021
https://gue.com/blog/building-a-fault-tolerant-rebreather-our-path-to-simplicity/

[Khlebnikov2023] Rostislav Khlebnikov, “Function Contracts in Practice,” *ACCU*, 2023

[Lakos15a] John Lakos, “Value Semantics: it Ain’t About the Syntax, Parts I, and II,” *CppCon*, 2015
https://www.youtube.com/watch?v=W3xI1HJUy7Q
https://www.youtube.com/watch?v=0EvSxHxFknM

[Lakos15b] John Lakos, “Value Semantics: it Ain’t About the Syntax," *ACCU*, 2015
https://accu.org/conf-docs/PDFs_2015/JohnLakos-Value%20Semantics.pdf

[Lakos20] John Lakos, *Large-Scale C++, Volume I, Process and Architecture*, Boston: Addison-Wesley, 2020

[Lakos22a] John Lakos, Vittorio Romeo, Alisdair Meredith, and Rostislav Khlebnikov, *Embracing Modern C++ Safely*, Boston: Addison-Wesley, 2022

[Lakos22b] John Lakos, "Embracing `noexcept` Operators and Specifiers Safely," *CppCon*, 2022
https://www.youtube.com/watch?v=VXPw1FJPhLA

[Liskov87] Barbara Liskov, "Data Abstraction and Hierarchy," OOPSLA, 1987
https://www.cs.tufts.edu/~nr/cs257/archive/barbara-liskov/data-abstraction-and-hierarchy.pdf

[Meyers15] Scott Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.* 1st edition. Sebastopol, CA: O'Reilly, 2015

[Mortoray13] Edaqa Mortoray, "The true cost of zero cost exceptions." Musing Mortoray Blog, September 12, 2013
https://mortoray.com/2013/09/12/the-true-cost-of-zero-cost-exceptions/

[O'Dwyer] Arthur O'Dwyer, "The Lakos Rule," *Stuff mostly about C++*, blog, April 25, 2018
https://quuxplusone.github.io/blog/2018/04/25/the-lakos-rule/

[Schoedl] Arno Schoedl, "A Practical Approach to Error Handling," *C++ on Sea*, 2022
https://www.youtube.com/watch?v=k7jjaS3FMWo