Slides for P2681R0:
Narrow Contracts and `noexcept`
Are Inherently Incompatible

Document #: P2949R0
Date: 2023-07-14
Project: Programming Language
Audience: EWG and LEWG
Reply-to: John Lakos
<jlakos@bloomberg.net>

Note: The following slides were presented by John Lakos in EWG on Friday, June 16, 2023, in Varna, Bulgaria, and reflect the essential ideas in P2861R0.

ABSTRACT

A contract is a plain-language specification of whatever essential behavior a given function promises to deliver when invoked in contract. A function that has at least one syntactically valid combination of state and input for which the behavior is undefined has a precondition and is therefore said to have a narrow contract. The Lakos Rule effectively prohibits placing the `noexcept` specifier (introduced in C++11) on any function that would otherwise have a narrow contract.

This talk begins with a reprise of contracts, essential behavior, and preconditions. It then contrasts two classic software design principles, Design by Contract and Liskov Substitutability, and uses the latter to explain how both backward compatibility and wide implementations benefit from scrupulously adhering to The Lakos Rule. We conclude that best practice is to follow this rule, especially in the specification of the C++ Standard Library, and we close with a welcome solution that satisfies essentially all needs and wants of the eclectic C++ multiverse.

Narrow Contracts and `noexcept` Are **Inherently Incompatible**

C++ Standards Committee

Varna, Bulgaria, June 16, 2023

John Lakos Senior Architect

TechAtBloomberg.com

P2861R0: *The Lakos Rule*

Objective for this presentation:

1. Demonstrate convincingly that ***The Lakos Rule*** is fundamentally sound software-engineering advice.
2. Observe that, apart from *move* operations, `noexcept` is unneeded in the Standard-Library specification.
3. Recommend appropriate use of `noexcept` in (1) the Standard Library, (2) conforming implementations, and (3) third-party or user libraries.

Contracts

Function Contract

- A bilateral agreement between a function's *implementor* and its (human) *client*
- Typically written in a plain (natural) language but not necessarily entirely so
- Represents (either explicitly or implicitly) any *preconditions* and clearly delineates **all** *essential behavior* promised when called *in contract*

Contracts

Function Contract (example):

```
int half(int x);
    // Return a value that is numerically half the
    // specified `x` value rounded toward zero.


double sqrt(double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

Function Contracts

Essential Behavior

- Comply with expressed post conditions.
    - The function always returns with a result.
        - The result is half of its input.
        - The result is the positive square root of its input.
- Honor any other promised behavior.
    - The function runs in constant (*O[1]*) time.
    - The function is *thread safe*.

Function Contracts

Implementation-Defined Behavior

- Behavior that is *not* specified, implied, or strongly suggested as being *essential* **within** the valid domain

```
struct Point { int x; int y; }

void mySort(Point *start, int length);
    // Sort the specified contiguous range of `Point`
    // objects in nondecreasing order of their
    // respective `x`-coordinate values, beginning at
    // the specified `start` address and extending for
    // (at least) the specified `length` objects.
```

Function Contracts

Implementation-Defined Behavior

```
void mySort(Point *start, int length);
    // Sort the specified contiguous range of `Point`
    // objects in nondecreasing order of their
    // respective `x` coordinate values, beginning at
    // the specified `start` address and extending for
    // (at least) the specified `length` objects.
```

Is there any room for implementation-defined behavior *within* the domain of this contract?

YES!

Function Contracts

Implementation-Defined Behavior

```
void mySort(Point *start, int length);
    // Sort the specified contiguous range of `Point`
    // objects in nondecreasing order of their
    // respective `x` coordinate values, beginning at
    // the specified `start` address and extending for
    // (at least) the specified `length` objects.

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

void f() { mySort(a, 3); }  // after we call `f`
```

Implementation-Defined Behavior

1. `a[]:  { { 8, 3 }, { 9, 2 }, { 9, 1 } }`
2. `a[]:  { { 8, 3 }, { 9, 1 }, { 9, 2 } }`

Function Contracts

Implementation-Defined Behavior

```
int half(int x);
    // Return a value that is numerically half the
    // specified `x` value rounded toward zero
```

Is there any room for implementation-defined behavior *within* the domain of this contract?

No

Function Contracts

Implementation-Defined Behavior

```
double sqrt(double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

Is there any room for implementation-defined behavior *within* the domain of this contract?

No*

*https://stackoverflow.com/questions/22546534/accuracy-of-sqrt-of-integers

Function Contracts

Implementation-Defined Behavior

```
float sqrt(long double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

Is there any room for implementation-defined behavior *within* the domain of this contract?

Yes* }

Let z^2 be the largest value for which z can be represented exactly as a `float`.
We can represent $9z^2$ exactly as a `long double`, but only $2z$ or $4z$ as a `float`.

*https://stackoverflow.com/questions/22546534/accuracy-of-sqrt-of-integers

Function Contracts

Implementation-Defined Behavior

```
float sqrt(long double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

OBSERVATION

The *declaration* of the function **informs** the contract.

(More on this later.)

Function Contracts

Preconditions

- What must be true of
  - *any* inputs
  - *all* relevant object (or program) state
- Otherwise, the behavior of invoking that function is *undefined.*
  - *Undefined behavior* is behavior for which there are **no** requirements.

Function Contracts

Preconditions *w.r.t.* `std::vector<T>:`

`vector()`	no
`std::size_t capacity() const;`	no
`void push_back(const T& v);`	no
`const T& front() const;`	yes
`T& operator[](std::size_t i);`	yes
`T& at(std::size_t i);`	no

Narrow Contracts and **`noexcept`** Are Inherently Incompatible

Wide Contracts

Preconditions *w.r.t.* `std::vector<T>:`

- `vector()` no
- `std::size_t capacity() const;` no
- `void push_back(const T& v>;` no
- `const T& front() const;` yes
- `T& operator[](std::size_t i);` yes
- `T& at(std::size_t i);` no

Narrow Contracts

Preconditions *w.r.t.* `std::vector<T>`:

`vector()`	no
`std::size_t capacity() const;`	no
`void push_back(const T& v>;`	no
`const T& front() const;`	yes
`T& operator[](std::size_t i);`	yes
`T& at(std::size_t i);`	no

Function Contracts

Explicit Preconditions

```
int half(int x);
    // Return a value that is numerically half the
    // specified `x` value rounded toward zero.
```

Does this function explicitly call out any preconditions?

No.

Function Contracts

Explicit Preconditions

```
double sqrt(double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

Does this function explicitly call out any preconditions?

Yes.

Importantly,
if we were to pass in **-1.0**, `sqrt`
doesn't need to return ***i*** (nor can it).

Function Contracts

Implicit Preconditions

```
double sqrt(double x);
    // Return a value whose representation is
    // numerically as close as possible to that of
    // the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

Does this function have any *implicit* preconditions?

- Yes. (But perhaps not what you think.)

What if we pass in a *NaN*?

- Nope, that's UB. `assert(0 <= NaN)` **// Fail!**

Function Contracts

Implicit Preconditions

```
int half(int x);
    // Return a value that is numerically half the
    // specified `x` value rounded toward zero.
```

Undefined behavior!!

Does this function have any *implicit* preconditions?

Yes. What if we pass in an *indeterminate value*?

```
int f() { int x; return half(x); }
```

Function Contracts

Implicit Preconditions

```
int half(int x);
    // Return a value that is numerically half the
    // specified `x` value rounded toward zero.  The
    // behavior is undefined if `x` has indeterminate value.
```

Does this function have any *implicit* preconditions?

e pass in an *indeterminate value*?

```
int x; return half(x); }
```

Really?

Function Contracts

Implicit Preconditions

```
void load(      int&  x) {  x = 0;   } // #0 None (UB if invalid)
void load(      int&& x) {  x = 0;   } // #1 None (bad idea)
void load(      int*  x) { *x = 0;   } // #2 Must point to an object

 int read(      int   x) { return x; } // #3 Must be initialized
 int read(      int&  x) { return x; } // #4 Must be initialized
 int read(const int&  x) { return x; } // #5 Must be initialized
 int read(      int&& x) { return x; } // #6 Must be initialized
 int read(const int&& x) { return x; } // #7 Must be initialized ??
 int read(      int*  x) { return x; } // #8 Must point to an ...
 int read(const int*  x) { return x; } // #9     ... initialized object
```

Function Contracts

Implicit Preconditions

OBSERVATION

Not *all* preconditions need to be stated explicitly.

Arguments that are to be

- *written* are required to be in a *constructed* state.
- *read* are required to be in an *initialized* state.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Declaration-Implied *Essential Behavior*

Does the declaration affect the contract?

- The function declaration provides the syntactic framework to which the plain-language contract refers:

```
double sqrt(double x);
    // This function (`sqrt`) takes a single argument (of
    // type `double`) and returns a value (of type `double)
    // that is the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

- Each parameter or return type is apparent.
- We may choose *not* to restate what is already codified.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Declaration-Implied *Essential Behavior*

Does the declaration affect the contract?

- The function declaration provides the syntactic framework to which the plain-language contract refers:

```
double sqrt(double x);
    // This function (`sqrt`) takes a single argument (of
    // type `double`) and returns a value (of type `double)
    // that is the positive square root of the specified `x`.
    // The behavior is undefined unless `0 <= x`.
```

- Any parameter or return types are apparent.
- We may choose *not* to restate what is already codified.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Declaration-Implied *Essential Behavior*

Does the declaration affect the contract?

- The function declaration provides the syntactic framework to which the plain-language contract refers:

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

- Any parameter or return types are apparent.
- We may choose *not* to restate what is already codified.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Declaration-Implied *Essential Behavior*

Does the declaration affect the contract?

- The function declaration provides the syntactic framework to which the plain-language contract refers:

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

Three-slashes comments define the contiguous syntactic element(s) below.

Suboptimal for development, but more tool friendly

- Any parameter or return types are apparent.
- We may choose *not* to restate what is already codified.

Function Contracts

Design by Contract (DbC) – Bertrand Meyer:

- An object of subtype D (of its supertype B) can be used in any context in which B could have been used and more.

But called via the base-class API?!

- Inheritance relationships must follow certain rules:
  - Derived preconditions must be a *superset* of those for the base.
  - Derived postconditions must be a *subset* of those for the base.
  - Importantly, postconditions result from the ***union*** of *all* input.
- The behavior **must** be *compatible* but **not** necessarily *identical.*
- His design principle applies to ***virtual*** functions ***only***.

Function Contracts

We'll talk more about *this topic* shortly.

Design by Contract (DbC) – Bertrand Meyer:

- An object of subtype D (of its supertype B) can be used in any context in which B could have been used and more.

Acts as a *wide implementation.*

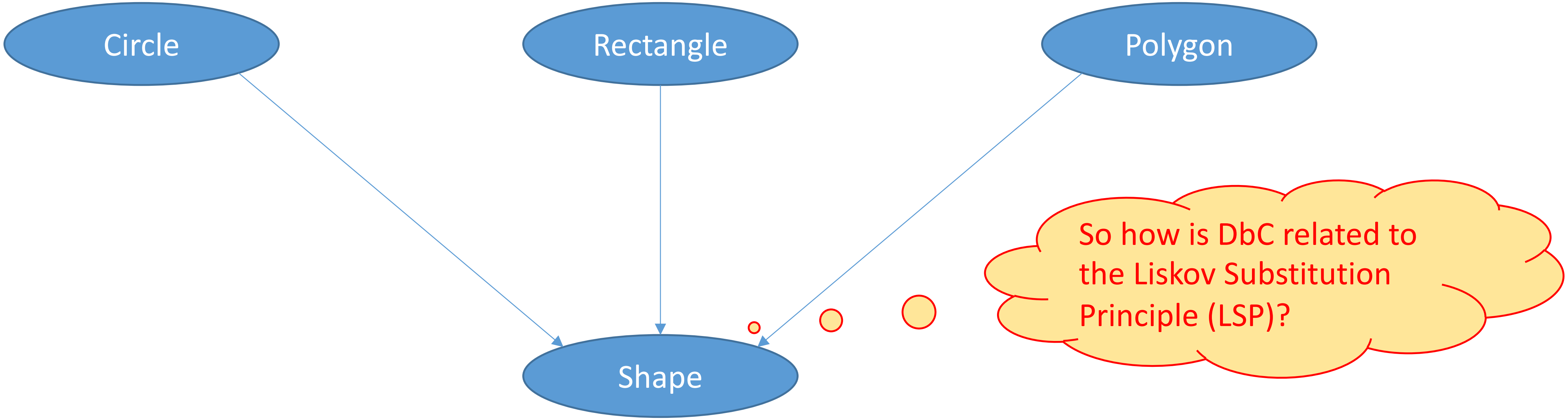
- Inheritance relationships must follow certain rules:
  - Derived preconditions must be *the same as* those for the base.
  - Derived postconditions must be a *subset* of those for the base.
  - Importantly, postconditions result from the ***union*** of *all* input.
- The behavior **must** be *compatible* but **not** necessarily *identical.*
- His design principle applies to ***virtual*** functions ***only***.

Design by Contract (DbC)

```
int numVertices() const
 [[ post r: 0 == r ]];
```

```
int numVertices() const
 [[ post r: 4 == r ]];
```

```
int numVertices() const
 [[ post r: 0 <= r ]];
```



Circle

Rectangle

Polygon

Shape

So how is DbC related to the Liskov Substitution Principle (LSP)?

```
virtual int numVertices() const = 0
          [[ post r: 0 <= r ]];
```

Design by Contract (DbC)

What is DbC good for?

- Heuristics for designing a sound hierarchy of polymorphic objects
  - Virtual functions support *variation in behavior*.*
- Should a C++ Contracts facility enforce it?
  - **Of course not!**
  - There are many valid reasons why one might deviate from these guidelines in practice.

The Standard Supports the Multiverse!

- We mention DbC *only* in contrast to our next topic.

* *C++ Programming Style* by **Tom Cargill** Paperback | Addison-Wesley Professional | Pub. Date: 1992-07-10. ISBN: 0201563657 | ISBN-13: 9780201563658.

Function Contracts

Liskov Substitutability (***not*** what “LSP” connotes)

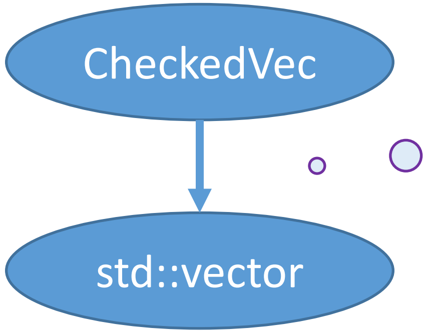
- An object of subtype `D` (of its supertype `B`) can be used in any context in which `B` could have been used and more:
  - The behavior for `D` in the domain of `B` is *(as-if)* identical.
  - The behaviors in `D` are ***not*** limited to those in B.
  - Importantly, the behaviors in `D` are *unconstrained* outside of the corresponding domain for `B`.
- Her design principle applies to ***non-virtual*** functions.
  - Concerns ***identical*** (not just ***similar***) behavior ***in contract***

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability (not “LSP”)

```
template <class T>
T& CheckedVec<T>::operator[](std::size_t index);
    // Return a reference to the element at the specified `index`
    // if `index < this->size()`; otherwise, throw `std::range_error`.
```

Nonvirtual Functions



Structural Inheritance

```
template <class T>
T& std::vector<T>::operator[](std::size_t index);
    // Return a reference to the element at the specified `index`.
    // The behavior is undefined unless `index < this->size()`.
```

We can use CheckedVec to catch accidental contract violations.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability Use Case

```
void load(std::vector<int>& v);   // Populate object with [ 2 1 0 4 3 ].
void sort(std::vector<int>& v);   // Note: Use of index is NOT checked in `sort`.

int main()  // Version 1.0
try {
    std::vector<int> v;               // Soon to be: CheckedVec<int> v;
    load(v);
    for (int i : v) { cout << v[i] << ' '; } cout << '\n';
    sort(v);                 <
    for (int i = 0; i <= v.size(); ++i) { cout << v[i] << ' '; } cout << '\n';
    return 0;
}
catch (std::exception& e) {
    std::cout << "Error: " << e.what() << '\n';
}
```

```
Output: 2 1 0 4 3
        0 1 2 3 4 8 ?
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability Use Case

```
void load(std::vector<int>& v);  // Populate object with [ 2 1 0 4 3 ].
void sort(std::vector<int>& v);  // Note: Use of index is NOT checked in `sort`.

int main()  // Version 1.0
try {
    CheckedVec<int> v;                   // Was: std::vector<int> v;
    load(v);
    for (int i : v) { cout << v[i] << ' '; } cout << '\n';
    sort(v);                   <
    for (int i = 0; i <= v.size(); ++i) { cout << v[i] << ' '; } cout << '\n';
    return 0;
}
catch (std::exception& e) {
    std::cout << "Error: " << e.what() << '\n';
}
```

With CheckedVec<int> we safely detect the contract violation (no more UB).

Output:

```
2 1 0 4 3
               ?
0 1 2 3 4 Error: bad index
```

Liskov Substitutability (not "LSP")

Why do we care about ***Liskov Substitutability*** ?!

- For the same reason we care about ***backward compatibility*** across software versions.
- Our goal has always been for any correct C++ program written to date to continue to work, with no observably different behavior, when built against future C++ Standards.

Liskov Substitutability (not "LSP")

The ***actual*** Liskov-Substitution Principle:

- If, for *all current programs** *P* written (correctly) in terms of the current version *V* of a library *L*, replacing *V* with *V+1 of L* results in ***no*** *change* in observable behavior for any *P*, then *V+1* is *substitutable* for *V*.

*In **theory**, we mean any program that *could be written* (e.g., by Machiavelli).

In **practice**, we mean one that *might occur* even accidently (e.g., by Murphy).

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and Versioning

```
void handler(int x);                    // version A1.0
    // Print the value of `x` to `stdout`.
    // Call `std::terminate`.
    // The behavior is undefined unless `1 <= x`.
```

Domain	Range
narrow	print terminate
1 <= x	

```
void handler(int x);                    // version A2.0
    // Print the value of `x` to `stdout`. If `x`
    // is positive, call `std::terminate`;
    // otherwise, throw `std::logic_error`.
    // The behavior is undefined unless `0 <= x`.
```

Domain	Range
narrow	print terminate
0 <= x	throw

```
void handler(int x);                    // version A3.0
    // Print the value of `x` to `stdout`. If `x`
    // is positive, call `std::terminate`; if `!x`,
    // throw `std::logic_error`; otherwise, return.
```

Domain	Range
wide	print terminate throw return

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and Versioning

Stable accumulation of client usage (P)

```
int f(int x)                      // (since version A1.0)
    // if `x >= 10`, return 10;
    // otherwise, print `1` and terminate.
{
    if (x >= 10) return 10;
    handler(1);
}
```

```
int g(int x)                      // (since version A2.0)
    // if `x >= 20`, return 20;
    // otherwise, print `0` and throw.
{
    if (x >= 20) return 20;
    handler(0);
}
```

```
int h(int x)                      // (since version A3.0)
    // if `x >= 30`, return 30;
    // otherwise, print `-1` and return 0.
{
    if (x >= 30) return 30;
    handler(-1); return 0;
}
```

Domain	Range
narrow **1 <= x**	print terminate
narrow **0 <= x**	print terminate throw
wide	print terminate throw return

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and Versioning

```
[[noreturn]] void handler(int x);   // version B1.0
    // Print the value of `x` to `stdout`.
    // Call `std::terminate`.
    // The behavior is undefined unless `1 <= x`.
```

```
[[noreturn]] void handler(int x);   // version B2.0
    // Print the value of `x` to `stdout`. If `x`
    // is positive, call `std::terminate`;
    // otherwise, throw `std::logic_error`.
    // The behavior is undefined unless `0 <= x`.
```

```
~~[[noreturn]]~~ void handler(int x); // version B3.0
    // Print the value of `x` to `stdout`. If `x`
    // is positive, call `std::terminate`; if `!x`,
    // throw `std::logic_error`; otherwise, return.
```

Domain	Range
narrow 1 <= x	print terminate
narrow 0 <= x	print terminate throw
wide	print terminate throw return

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and Versioning

Stable accumulation of client usage (P)

```
int f(int x)                                // (since version A1.0)
    // if `x >= 10`, return 10;
    // otherwise, print `1` and terminate.
{
    if (x >= 10) return 10;
    [[noreturn]] handler(1);
}
```

Does not exist…

```
int g(int x)                                // (since version A2.0)
    // if `x >= 20`. return 20;
    // otherwise, print `0` and throw.
{
    if (x >= 20) return 20;
    [[noreturn]] handler(0);
}
```

…but it probably should!

```
int h(int x)                                // (since version A3.0)
    // if `x >= 30`, return 30;
    // otherwise, print `-1` and return 0.
{
    if (x >= 30) return 30;
    handler(-1); return 0;
}
```

Domain	Range
narrow **1 <= x**	print terminate
narrow **0 <= x**	print terminate throw
wide	print terminate throw return

Liskov Substitutability and Versioning

slightly
better
codegen

Consider these two contracts:

```
                                                      Domain      Range
void handler(int x);                  // version A1.0
    // Print the value of `x` to `stdout`.            narrow      print
    // This function does not return.                             noreturn
    // The behavior is undefined unless `1 <= x`.     1 <= x
```

```
[[noreturn]] void handler(int x);     // version B1.0
    // Print the value of `x` to `stdout`.            narrow      print
    // The behavior is undefined unless `1 <= x`.     1 <= x      noreturn
```

Which is *Liskov substitutable* for the other?

That is, which one is usable in a (proper) superset of situations for which the other one is ideal?

Function Contracts (terminology)

Implementation Contracts

- Implied Contract
  - of an implementation
- Conforming Implementation
  - of a (public) interface/contract
- Wide Implementation
  - of a (public) interface/contract

Function-Implementation Contracts

Implied Contract (of an implementation)

- The *implied contract* of a function is the envelope of *defined behavior* that can be gleaned from its declaration and implementation, including any information contained in the public contracts of any functions used in its implementation.

Implied Contract (of an implementation)

Implementation:

```
int half(int a) {
    return a / 2;
}
```

Interface + implied contract:

```
int half(int a);
    // Return half the specified `value` rounded
    // toward zero.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Implied Contract (of an implementation)

Implementation:

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

Interface + implied contract:

```
double sqrt(double value);
    // Return the positive square root of the specified
    // `value`.  The behavior is implementation defined
    // unless `value >= 0`.  Note that this function
    // will return a `NaN`, if supported, when given a
    // negative `value`.
```

Function-Implementation Contracts

Conforming Implementation (of an interface)

- If the *implied contract* of an implementation (subtype) `.c` — with respect to the public contract delineated by its interface (supertype) `.h` — satisfies `.h`'s contract in every context in which `.h` can be used *in contract* (and perhaps more), then `.c` is a *conforming implementation* of `.h`.

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {

    return (a + b)/2;
}
```

NO!

(Can overflow)

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {

    assert(a <= b);

    return a + (b - a)/2;
}
```

NO!

(Incorrect for A > B)

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {

    if (a > b) swap(a, b);     // assert(a <= b)

    return a/2 + (b - a)/2;
}
```

No!

(Incorrect for negative values)

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {
    if (a > b) std::swap(a, b);  assert(a <= b);
    if (a >= 0)      return a + (b - a) / 2;
    else if (b <= 0) return b + (a - b) / 2;
    else             return (a + b)/ 2;
}
```

Yes!

Narrow Contracts and `noexcept` Are Inherently Incompatible

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {
    int r = a / 2 + b / 2;
    int h = a % 2 + b % 2;
    if (h/2)                                   r += h/2;
    else if (r > 0 && h < 0 || r < 0 && h > 0) r += h;
    return r;
}
```

Yes!

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {

    return (static_cast<long long>(a) + b)/2;

    static_assert(sizeof(long long) > sizeof(int));
}
```

Yes!

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {
    return std::midpoint(a, b);
}
```

NO!

Rounding isn't toward 0

Interface + Contract:

```
template <class T>
T std::midpoint::average(T a, T b);
    // Return half the sum of `a` and `b`. No overflow occurs. If `a` and `b`
    // have integer type and the sum is odd, the result is rounded toward `a`.
```

Narrow Contracts and `b` Are Inherently Incompatible

Conforming Implementation (of an interface)

Interface + Contract:

```
int average(int a, int b);
    // Return the midpoint between the specified
    // `a` and `b` values, rounded toward 0.
```

Implementation of above; is it conforming?

```
int average(int a, int b) {
    int r = std::midpoint(a, b);
    if (a < b) { if (r < 0) r += (a ^ b) & 1; }
    else       { if (r > 0) r -= (a ^ b) & 1; }
    return r;
}
```

Yes!

Function-Implementation Contracts

Wide Implementation (of a [narrow] interface)

- If the *implied contract* of an implementation, `.c` , is (1) *conforming* and (2) offers a wide(r) *usable domain* (e.g., no preconditions) than that of its interface, `.h`, (i.e., having a narrow contract) we refer to `.c` as a *wide(r) implementation*.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

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

Yes!

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementation (of an interface)

Interface + Contract:

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

~~Wide~~ (conforming) implementation:

```
double sqrt(double value) {
    [[assume 0 <= value]]
    return std::sqrt(value);
}
```

Calling any C++ Standard Library function out of contract today is (language) undefined behavior.

Not so for any other functions and need not be so for Standard-Library implementations!

We should talk about this issue more later.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

```
double sqrt(double value) {
    if (value < 0) return -1;
    return std::sqrt(value);
}
```

Yes!

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

```
double sqrt(double value) {
    if (value < 0) throw std::logic_error;
    return std::sqrt(value);
}
```

Yes!

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

```
double sqrt(double value) {
    assert(value >= 0);
    return std::sqrt(value);
}
```

Yes!

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

```
double sqrt(double value) {
    [[assert: value >= 0]];
    return std::sqrt(value);
}
```

Contracts
Attribute
Notation

Yes!

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementation (of an interface)

Interface + Contract:

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

Wide (conforming) implementation:

```
double sqrt(double value) [[ pre: value >= 0 ]]
{
    return std::sqrt(value);
}
```

Contracts
Attribute
Notation

Yes!

Function Contracts and noexcept

Argument values are irrelevant!

The noexcept specifier

- Ensures that a function does not throw.
  - Often connotes that a function does not *fail*.
- void f(int x);
  - static_assert(false == noexcept(f(*(int*)(0))));
- void g(int x) noexcept;
  - static_assert(true == noexcept(g(*(int*)(0))));

operator, not specifier

specifier

unevaluated operands

Narrow Contracts and **`noexcept`** Are Inherently Incompatible

Implied Contract (of an implementation)

Implementation:

```
double sqrt(double value) {
    if (value < 0) throw std::logic_error(“negative”);
    return std::sqrt(value);
}
```

Interface + implied contract:

```
double sqrt(double value);
    // Return the positive square root of the
    // specified `value` if `value >= 0`; otherwise,
    // throw `std::logic_error(“negative”)`. Note that
    // `noexcept(sqrt(x))` is `false` for all `x`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Implied Contract (of an implementation)

Implementation:

```
double sqrt(double value) {
    if (value < 0) return 0.0;
    return std::sqrt(value);
}
```

Interface + implied contract:

```
double sqrt(double value);
    // Return the positive square root of the
    // specified `value` if `value >= 0`; otherwise,
    // return 0.  Throws nothing and `noexcept(sqrt(x))`
    // is `false` for all `x`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Implied Contract (of an implementation)

Implementation:

```
double sqrt(double value) noexcept {
    if (value < 0) return 0.0;
    return std::sqrt(value);
}
```

Interface + implied contract:

```
double sqrt(double value);
    // Return the positive square root of the
    // specified `value` if `value >= 0`; otherwise,
    // return 0.  Throws nothing and `noexcept(sqrt(x))`
    // is `true` for all `x`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Implied Contract (of an implementation)

Implementation:

```
double sqrt(double value) noexcept {
    if (value < 0) throw std::logic_error(“negative”);
    return std::sqrt(value);
}
```

Interface + implied contract:

```
double sqrt(double value);
    // Return the positive square root of the
    // specified `value` if `value >= 0`; otherwise,
    // call `std::terminate()`.  Throws nothing and
    // `noexcept(sqrt(x))` is `true` for all `x`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

```
                                                             Domain    Range
T& operator[](std::size_t i);                  // version A1.0
    // Return the element at the specified `i` position.     narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.      i < size()


T& operator[](std::size_t i);                  // version A2.0
    // Return the element at the specified `i` position.     narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i <= size()`.     i <= size()


T& operator[](std::size_t i);                   // version A3.0
    // Return the element at the specified `i` position       wide     return
    // if `i <= size`; otherwise, throw `std::range_error()`.          throw
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Stable accumulation of client usage (P)

```
T f(C<T>& c, std::size_t j)                 // (since version A1.0)
    // Return `c[j]`.
    // The behavior is undefined unless `j < c.size()`.
{
    return c[j];
}
```

```
T f(C<T>& c, std::size_t j)                 // (since version A2.0)
    // Return `c[j]`.
    // The behavior is undefined unless `j <= c.size()`.
{
    return c[j];
}
```

```
T f(C<T>& c, std::size_t j)                 // (since version A3.0)
    // If `j <= c.size()` return c[j]; otherwise, throw something.
{
    return c[j];
}
```

Domain	Range
narrow **i < size()**	return
narrow **i <= size()**	return
wide	return throw

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

```
                                                               Domain    Range
T& operator[](std::size_t i) noexcept;     // version B1.0               noexcept
    // Return the element at the specified `i` position.       narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.        i < size()


T& operator[](std::size_t i) noexcept;     // version B2.0               noexcept
    // Return the element at the specified `i` position.       narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i <= size()`.       i <= size()


T& operator[](std::size_t i) noexcept;     // version B3.0               noexcept
    // Return the element at the specified `i` position         wide     return
    // if `i <= size`; otherwise, ~~throw `std::range_error()`.            throw~~
                                  call `std::terminate()`.               terminate
```

Liskov Substitutability and `noexcept`

Consider these two contracts:

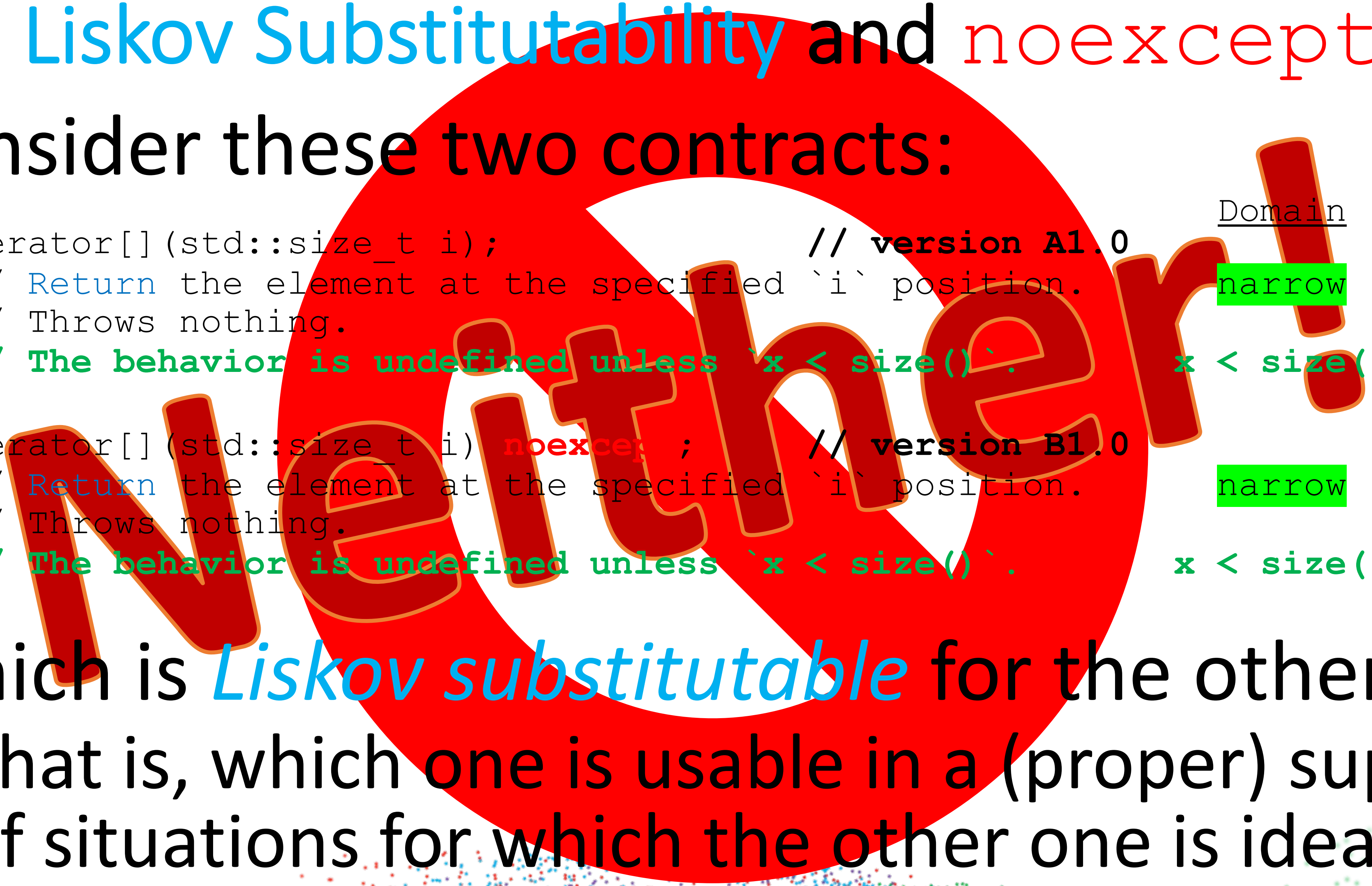
```
                                                              Domain    Range
T& operator[](std::size_t i);              // version A1.0
    // Return the element at the specified `i` position.      narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.      i < size()

T& operator[](std::size_t i) noexcept;     // version B1.0              noexcept
    // Return the element at the specified `i` position.      narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.      i < size()
```

Which is *Liskov substitutable* for the other?

That is, which one is usable in a (proper) superset of situations for which the other one is ideal.

Liskov Substitutability and `noexcept`

Consider these two contracts:



```
                                                                   Domain     Range
T& operator[](std::size_t i);                 // version A1.0
    // Return the element at the specified `i` position.           narrow     return
    // Throws nothing.
    // The behavior is undefined unless `x < size()`.           x < size()

T& operator[](std::size_t i) noexcept;        // version B1.0                 noexcept
    // Return the element at the specified `i` position.           narrow     return
    // Throws nothing.
    // The behavior is undefined unless `x < size()`.           x < size()
```

Which is *Liskov substitutable* for the other?

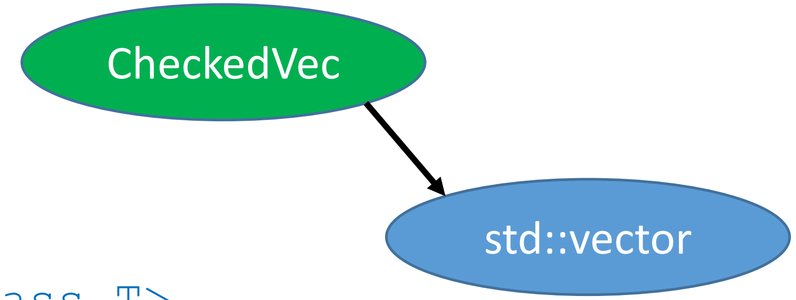
That is, which one is usable in a (proper) superset of situations for which the other one is ideal.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

To see why, consider two similar checked vectors, each derived *structurally* from `std::vector`:

```
template <class T>
T& CheckedVec<T>::operator[] (std::size_t index);
    // Return a reference to the element at the specified `index`
    // if `index < this->size()`; otherwise, throw `std::range_error`.
```



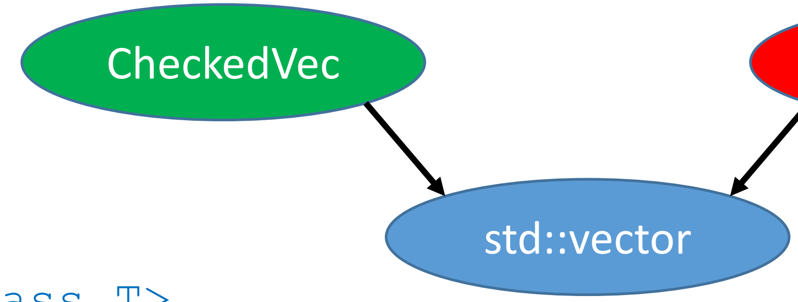
```
template <class T>
T& std::vector<T>::operator[](std::size_t index);
    // Return a reference to the element at the specified `index`.
    // The behavior is undefined unless `index < this->size()`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

To see why, consider two similar checked vectors, each derived *structurally* from `std::vector`:

```
template <class T>
T& CheckedVek<T>::operator[](std::size_t index) noexcept;
    // Return a reference to the element at the specified `index`
    // if `index < this->size()`; otherwise, ~~throw `std::range_error`~~.
                                          call `std::terminate`
```



CheckedVec

CheckedVek

std::vector

```
template <class T>
T& std::vector<T>::operator[](std::size_t index);
    // Return a reference to the element at the specified `index`.
    // The behavior is undefined unless `index < this->size()`.
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Checked vector with *throwing* `operator[]:`

```
template<class T>
struct CheckedVec : std::vector<T>
{
    using std::vector<T>::vector;  // inheriting constructors
    T& operator[](std::size_t index) {
        std::cout << “[CheckedVec] " << std::flush;
        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);
    }
};
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Checked vector with ***non**throwing* `operator[]`:

```
template<class T>
struct CheckedVek : std::vector<T>
{
    using std::vector<T>::vector;  // inheriting constructors
    T& operator[](std::size t index) noexcept {
        std::cout << “[CheckedVek] " << std::flush;
        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);
    }
};
```

Liskov Substitutability and `noexcept`

Then, consider this *generic* **`lookup`** function:

Without contracts, bugs are features.

```
template <typename C>
typename C::value_type& lookup(C& c, std::size_t i);
    // Using the bracket operator for the specified container, `c`, return
    // the element at the specified index, `i`, unless `i >= c.size()`.  If
    // `noexcept(c[i]) == true` and `i >= c.size()` throw `std::logic_error`;
    // otherwise, the behavior is whatever is defined (in the current build
    // mode) for `c`’s `noexcept(false)` non-`const` `operator[]`.
{
    if constexpr ( noexcept(c[i]) )  // Bracket operator is `noexcept(true)`.
    {
        if (i > c.size()) throw std::logic_error("BAD INDEX");
    }
    return c[i];  // If `i > c.size()` then `noexcept(c[i]) == false`.
}
```

For `i > size`, `operator[]` is called *iff* it is `noexcept(false)`.

Liskov Substitutability and `noexcept`

Now, consider this main program, which calls `lookup`:

```
int main() {
    const int init = 0xDeadBeef;  int ret = init;
    try {
        CheckedVec<int> vec;  // Bracket operator is `noexcept(false)`.
        ret = lookup(vec, 0);
    }
    catch (...) { std::cout << "Caught `noexcept(false)`.\n"; }
    try {
        CheckedVek<int> vek;  // Bracket operator is `noexcept(true)`.
        ret = lookup(vek, 0);
    }
    catch (...) { std::cout << "Caught `noexcept(true)`.\n"; }
    assert(ret == init);  return 0;  // status
}
```

`vec` is *empty.*
`vec.size() == 0`

`vek` is *empty*

```
[CheckedVec] Caught `noexcept(false)`.
[CheckedVek] terminate called after throwing an instance of 'std::range_error'
  what():  bad index
Aborted (core dumped) Status = -1
```

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Let’s take another look at our generic `lookup` function:

```
template <typename C>
typename C::value_type& lookup(C& c, std::size_t i);
    // Using the bracket operator for the specified container, `c`, return
    // the element at the specified index, `i`, unless `i >= c.size()`.  If
    // `noexcept(c[i]) == true` and `i >= c.size()` throw `std::logic_error`;
    // otherwise, the behavior is whatever is defined (in the current build
    // mode) for `c`’s `noexcept(false)` non-`const` `operator[]`.
{
    if constexpr ( noexcept(c[i]) )  // Bracket operator is `noexcept(true)`.
    {
        if (i > c.size()) throw std::logic_error("BAD INDEX");
    }
    return c[i];  // If `i > c.size()` then `noexcept(c[i]) == false`.
}
```

If `i == c.size()` then `noexcept(c[i])` *might* be `true`!

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Let’s take another look at our *generic* **`lookup`** function:

```
template <typename C>
typename C::value_type& lookup(C& c, std::size_t i);
    // Using the bracket operator for the specified container, `c`, return
    // the element at the specified index, `i`, unless `i >= c.size()`.  If
    // `noexcept(c[i]) == true` and `i >= c.size()` throw `std::logic_error`;
    // otherwise, the behavior is whatever is defined (in the current build
    // mode) for `c`’s `noexcept(false)` non-`const` `operator[]`.
{
    if constexpr ( noexcept(c[i]) )  // Bracket operator is `noexcept(true)`.
    {
        if (i >= c.size()) throw std::logic_error("BAD INDEX");
    }
    return c[i];  // If `i >= c.size()` then `noexcept(c[i]) == false`.
}
```

~~**If `` `i == c.size()` `` then `` `noexcept(c[i])` `` *might* be `` `true` ``!**~~

For `i >= size`, `operator[]` is called *iff* it is `noexcept(false)`.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Liskov Substitutability and `noexcept`

Now, consider this main program, which calls **`lookup`**:

```
int main() {
    const int init = 0xDeadBeef;  int ret = init;
    try {
        CheckedVec<int> vec;   // Bracket operator is `noexcept(false)`.
        ret = lookup(vec, 0);
    }
    catch (...) { std::cout << "Caught `noexcept(false)`.\n"; }
    try {
        CheckedVek<int> vek;   // Bracket operator is `noexcept(true)`.
        ret = lookup(vek, 0);
    }
    catch (...) { std::cout << "Caught `noexcept(true)`.\n"; }
    assert(ret == init);  return 0;  // status
}
```

```
[CheckedVec] Caught `noexcept(false)`.
Caught `noexcept(true)`.   Status = 0
```

} ***Distinct <u>Essential</u> Behaviors!***

For `i >= size`, `operator[]` is called *iff* it is `noexcept(false)`.

Liskov Substitutability/Backward Compatibility

Backward compatibility is *subjective*:

A. **Pure Liskov** – can’t write a program that would break.
   - E.g., even unspecified object size (introspection) doesn’t change.

B. **Applied Liskov** – wouldn’t write one that would break.
   - E.g., parsing `>>` as individual tokens for nested templates.

C. **Backward Compatible** – good enough for C++ Standard.
   - E.g., adding a keyword, such as `noexcept` or `co_return`.

D. **Incompatible** – inherently conflicting *essential* behavior.
   - E.g., when variables defined in a `for` statement became *local*.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Backward Compatibility and `noexcept`

Let's take one more look at these two contracts:

```
                                                              Domain    Range
T& operator[](std::size_t i);              // version A1.0
    // Return the element at the specified `i` position.      narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.      i < size()

T& operator[](std::size_t i) noexcept;     // version B1.0              noexcept
    // Return the element at the specified `i` position.      narrow    return
    // Throws nothing.
    // The behavior is undefined unless `i < size()`.      i < size()
```

Why?

Neither is *Liskov substitutable* for the other!

Which is *backward compatible* with the other?

Backward Compatibility and `noexcept`

Why *adding* `noexcept` is backward compatible

- Either *adding* or *removing* `noexcept` *might* affect the behavior of an arbitrary client in arbitrary ways.
- *Adding* `noexcept` to a function ***that doesn't throw*** should — if anything — act as a pure optimization in practice.
- *Removing* `noexcept` might act as a pessimization.
  - E.g., from a *move* or *copy* constructor could result in slower *copy* algorithm to preserve the *strong exception-safety guarantee*.
- ***Only*** if *generic* client uses `noexcept` *operator* on the function!
  - Otherwise, the function's object code might be *larger* but **not** *faster*.

Backward Compatibility and `noexcept`

Why *adding* `noexcept` is backward compatible

```
Template <typename F, int x>
void f(F x) {
    if constexpr (noexcept(F(x)) {  // fast algo
        // ... (cannot throw)
    }
    else {                          // slow algo
        // ...
        // ... (might throw)
        // ...
    }
}
```

Move Operations *ONLY!*

Narrow Contracts and `noexcept` Are Inherently Incompatible

The "need" for `noexcept`

A. **Declaring non-throwing move operations**
   - The *raison d'être* of the `noexcept` *specifier* (and *operator*).

B. **Wrapper redeclaring move operations**
   - A practical way to improve performance based on *global knowledge*.

C. **Callback framework directly supporting** noexcept **functions**
   - Easy alternatives are to provide (1) a default or (2) nonthrowing wrapper.

D. **Enforce explicit documentation**
   - A simple alternative is to document the function as "nonthrowing."

E. **Reduce object-code size**
   - An often-preferable alternative is to build with exceptions disabled.

F. **Unrealizable runtime-performance benefits**
   - The ***zero-cost exception model*** renders any such effort ***futile*** in practice.

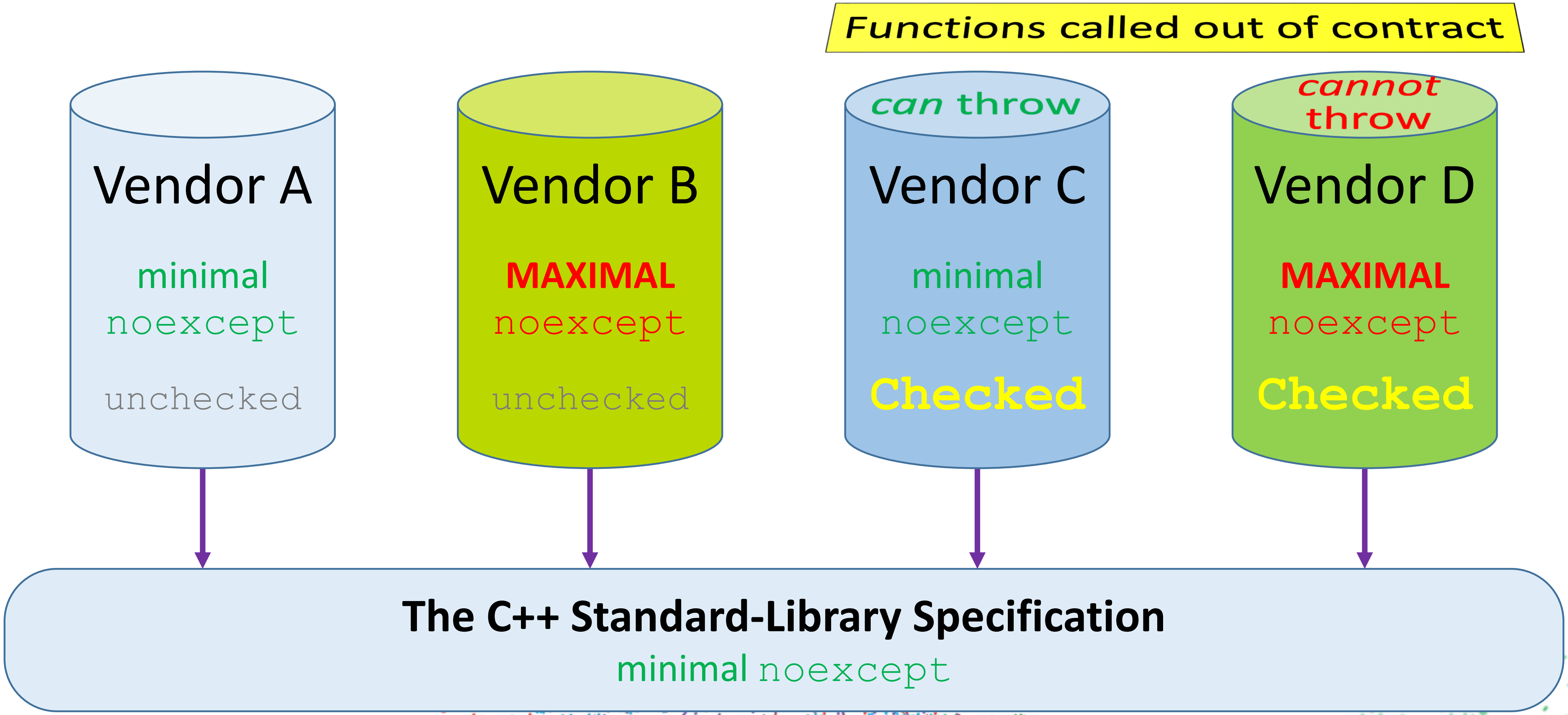
The *trouble* with `noexcept`

Why *adding* `noexcept` can be problematic

- **Accidental termination**
  - Having more functions than necessary declared `noexcept` doesn’t help matters.
  - Especially for those who make use of exceptions.
- **Incompatibility with *narrow contracts***
  - Precludes ***wide implementations*** that might throw.
  - *Critical* for the **C++ Standard-Library *Specification***.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementations of the C++ Standard Library



Functions called out of contract

Vendor A
minimal `noexcept`
`unchecked`

Vendor B
MAXIMAL `noexcept`
`unchecked`

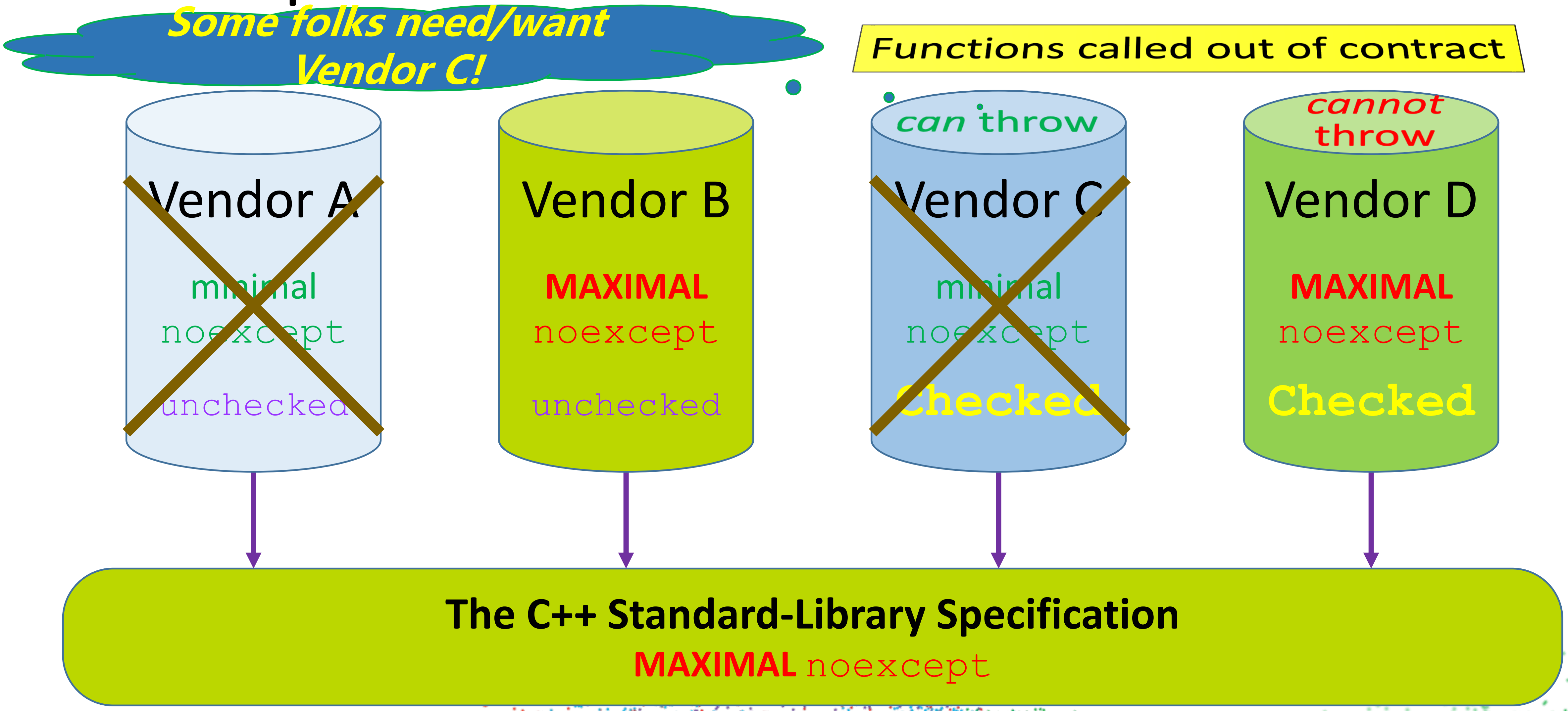
can throw
Vendor C
minimal `noexcept`
Checked

cannot throw
Vendor D
MAXIMAL `noexcept`
Checked

The C++ Standard-Library Specification
minimal `noexcept`

Narrow Contracts and `noexcept` Are Inherently Incompatible

Wide Implementations of the C++ Standard Library



Some folks need/want Vendor C!

Functions called out of contract

~~Vendor A~~
~~minimal~~
~~`noexcept`~~
~~`unchecked`~~

Vendor B
MAXIMAL
`noexcept`
`unchecked`

can throw
~~Vendor C~~
~~minimal~~
~~`noexcept`~~
~~**Checked**~~

cannot throw
Vendor D
MAXIMAL
`noexcept`
Checked

The C++ Standard-Library Specification
MAXIMAL `noexcept`

Narrow Contracts and `noexcept` Are Inherently Incompatible

Our universe versus *The Multiverse*

What happens when a logic defect is detected?

☐ Terminate immediately.
☐ Save client data, release resources, and terminate.
☐ Signal an error and then block or busy wait.
☐ Log a diagnostic, continue, and hope for the best.
☐ Snapshot, then throw `std::logic_error`.
☐ Throw some other kind of object.

Narrow Contracts and `noexcept` Are Inherently Incompatible

Our universe versus *The Multiverse*

What happens when a logic defect is detected?

- ☐ Terminate immediately.
- ☐ Save client data, release resources, and terminate.
- ☐ Signal an error and then block or busy wait.
- ☐ Log a diagnostic, continue, and hope for the best.
- ☐ Snapshot, then throw `std::logic_error`.
- ☐ Throw some other kind of object.

Any of these might be optimal, depending on the

- industry
- organization
- application

Conclusion

Barbara Liskov
is a Rockstar!

Liskov Substitutability is the goal!

Liskov Substitutability ➔ Backward Compatibility

But not vice versa!

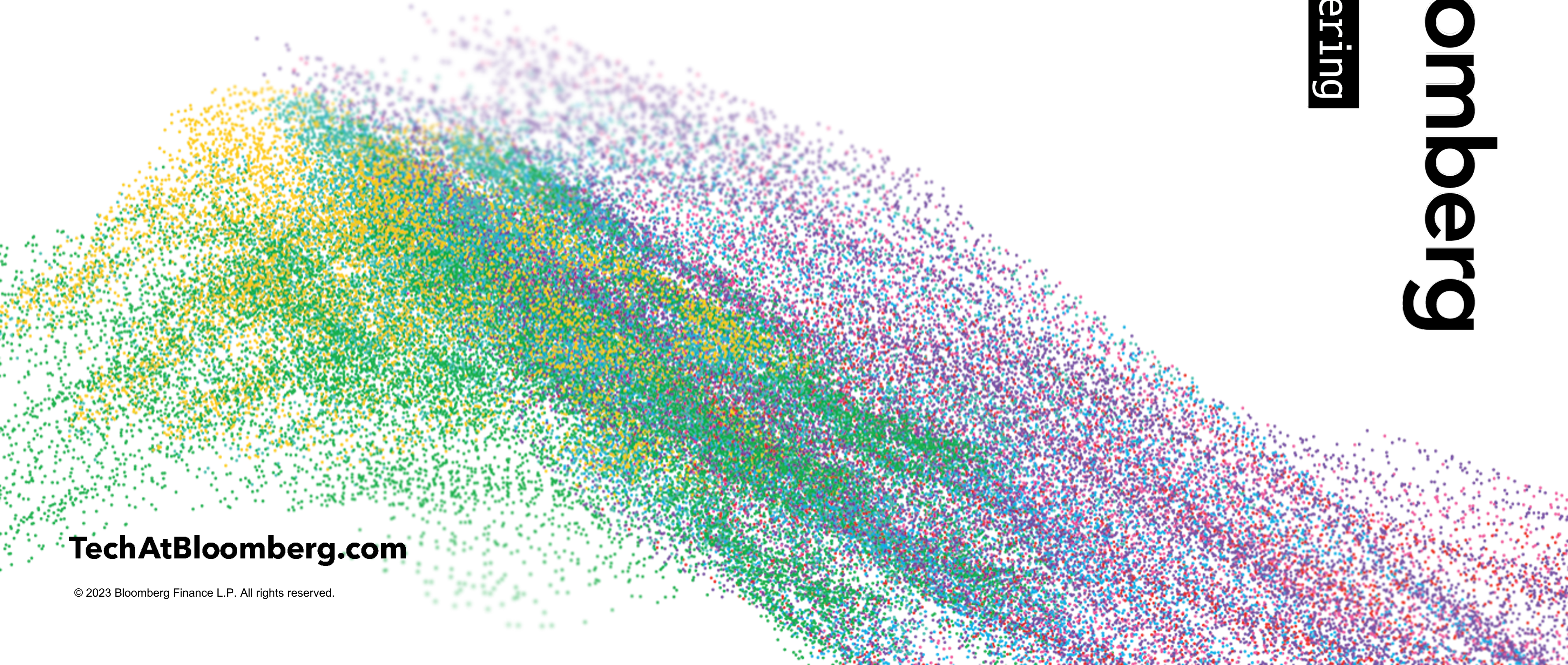
Conclusion

The C++ Standard is for the *multiverse!*

1. ***Never* require the `noexcept` specifier on *any* standard function** *unless* effective use of that function might reasonably require (direct or indirect) use of the `noexcept` operator (e.g., from a generic context) — i.e., *move* operations only.
2. **Allow implementations to strengthen exception specifications (i.e., add `noexcept` specifiers)** *unless* a function's contract is (1) narrow or (2) involves callbacks that have narrow contracts or might throw in contract.

Thank you!

Bloomberg Engineering

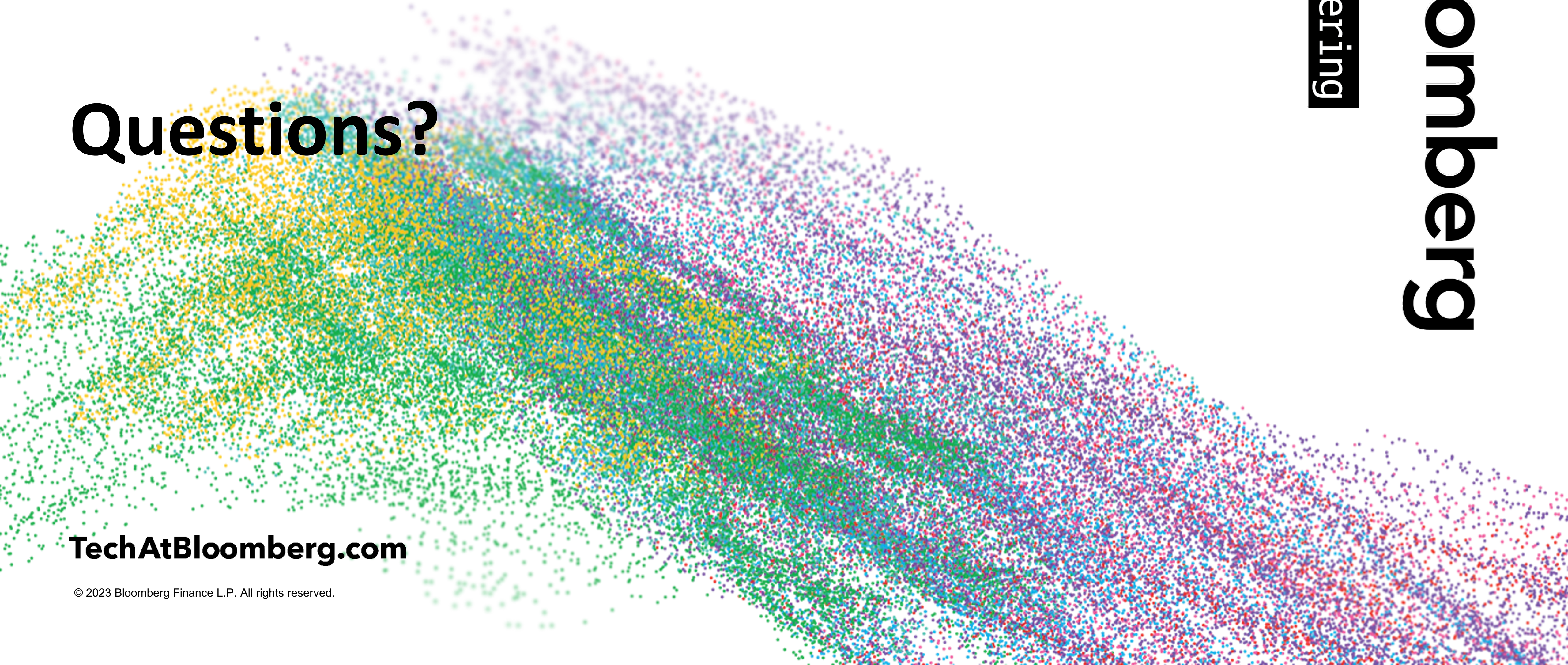


TechAtBloomberg.com

We are hiring!

https://www.bloomberg.com/careers

Questions?



TechAtBloomberg.com

Bloomberg Engineering