Noexcept?

Enabling Testing of Contract Checks

PABLO HALPERN
& TIMUR DOUMLER

Cppcon
The C++ Conference

2023
October 01 - 06

Welcome

What can the Screen Actors Guild and the Writers Guild of America tell us about software development?

- Good programs require good contracts.
- Corollary: Good contracts require good enforcement.
  - Contract checks help catch bugs early in the process.
- Observation: Good enforcement requires good oversight.
  - **Contract checks should themselves be tested.**

Today's talk

Goals – What We Hope to Teach You

- How the `noexcept` specifier and `noexcept` operator work and what they are for
- The importance of writing contracts and checking them at run time
- How to unit test contract-checking annotations (CCAs)
- How `noexcept` interacts with unit testing of CCAs.

We will go over everything twice!

About the presenters

In Absentia

Pablo Halpern

- Lives in Boston, MA, USA
- Member of the C++ Standards Committee; Library Evolution WG
- Contributor to *Embracing Modern C++ Safely* (Pearson 2021)
- Interests: Allocators, parallelism, best practices
- Most recent joint replacement: left hip

Timur Doumler

- Lives in Espoo Finland
- Member of the C++ Standards Committee; chair of the contracts SG
- Co-host of CppCast
- Organizer of the Helsinki C++ Meetup
- Interests: Clean code, good tools, low latency, and the evolution of C++.

A Quick Review of the `noexcept` Language Feature

We'll get back to contract talks soon

The `noexcept` *Specifier*

- When used to decorate a function declaration, `noexcept` indicates that the function will not throw an exception:

```
bool is_numeric(const std::string&) noexcept;
MyClass(MyClass&&) noexcept;                 // Move constructor
~MyClass();                                  // destructors are implicitly noexcept.
```

- Callers do not need to defend against possible exceptions:

```
int *array = new int[N];   // Possible leak hazard
if (is_numeric(s)) …       // does not throw
…
delete[] array;            // Safe, no leak because no exception can occur
```

Conditional `noexcept` Specifier

- A (compile-time) constant expression can be supplied to a `noexcept` specifier. If `true`, the function is `noexcept`; if `false`, it is not:

```
void f1() noexcept(true);          //  won't throw – same as void f1() noexcept;
void f2() noexcept(false);         //  might throw – same as void f2();
~SomeClass() noexcept(false);      //  might throw – only way to write a throwing destructor
```

- Conditional `noexcept` is useful when the exception specification depends on a compile-time trait of a template parameter:

```
template <class T>
T* ObjectResource::allocate() noexcept(sizeof(T) <= SmallObjectSize);
    //  Allocate from small object buffer if T fits (no-throw), else from general heap (might throw)
```

Throwing in a `noexcept` Context

- If an exception attempts to *escape* a `noexcept` function, the program ends via a call to `std::terminate`:

```
bool is_numeric(const std::string& s) noexcept {
  if (s.size() > 300)
    throw LengthError();   // terminate if s is too long
  …
```

- A `throw` expression is OK, so long as it is never invoked:

```
bool x = is_numeric("hello");   // OK, does not attempt to throw,
std::string longstr(350, 'x'); // Create a long string.
bool y = is_numeric(longstr);   // terminate – attempted throw from noexcept
```

The `noexcept` *Operator*

- The `noexcept` operator yields a compile-time Boolean value indicating whether an expression is allowed to emit an exception:

```
int a, b;  vector<int> v;
static_assert(noexcept(a + b));             //  OK, int addition can't throw
static_assert(noexcept(v.empty()));         //  OK, vector::empty can't throw
static_assert(noexcept(v.push_back(a)));  //  ERROR: push_back can throw
```

- As in `sizeof(`*expr*`)`, the expression in `noexcept(`*expr*`)` is not evaluated.
- A `true` result is typically used at compile time to enable an algorithm that is more performant but would be unsafe in the presence of exceptions.

Contracts and Contract Checking

I told you we'd get back to contracts!

What is a Contract?

- In English, a contract is an agreement of the form “I promise to do this if you promise to do that.”
- In software, a *function contract* is
  - A set of *preconditions* that the caller agrees to meet
  - A set of *postconditions* that the function agrees to meet, provided the preconditions have been met
- Example contract for `vector::pop_back()` :
  - Precondition: the `vector` is not empty
  - Postcondition: the last element of the `vector` has been removed

Wide versus Narrow Contracts

- A function has a *wide contract* if it has no preconditions:
  - Example: unsigned integer addition
  - Example: `vector::size()`
- A function has a *narrow contract* if it has preconditions:
  - Example: signed integer addition (precondition: a + b will not over/underflow)
  - Example: `vector::front()` (precondition: the `vector` is not empty)
- When a precondition is violated, the result is *Undefined Behavior*
  - Anything can happen, including crashing or corrupting data
  - Postconditions do not have to hold – the contract is broken

Contract Violations

- Failure to meet a precondition or postcondition is a **bug** in the code.
- Catching contract violations early results in fewer bugs in released software.
- Some preconditions and postconditions can be expressed as code and checked for compliance; others cannot.
  - "The input pointer is not null" is easily expressed and checked in code.
  - "The input array is sorted" can be expressed in code, but evaluation is probably too expensive to check.
  - "The output is a random number" *cannot* easily be expressed in code but it *is* part of the human-language documentation.

Checking for Contract Violations

- Many contract violations can be detected in code:

```
template <class T, class Alloc>
void vector<T, Alloc>::pop_back() {
  size_type sz = this->size();
  assert(sz > 0);               // Precondition check
  m_data[sz - 1].~T();          // Destroy last element
  assert(size() == sz - 1);     // Postcondition check
}

vector<int> v;
v.pop_back();  // ASSERTION FAILURE: Precondition violation
```

Postcondition will always fail!

Contract-checking Annotations (CCAs)

- The C `assert` macro in the `<cassert>` header
- Homebrew or 3rd party macro libraries
  - `PRECONDITION(cond)`, `POSTCONDITION(cond)`
  - `ASSERT(cond)`
- Upcoming C++ contract annotations (study group 21)
  - `[[pre:` *cond*`]]`, `[[post:` *cond*`]]` or `pre{`*cond*`}`, `post{`*cond*`}`
  - `[[assert:` *cond*`]]` or `assert {`*cond* `}`

Unit Testing, CCAs and Exception Specifications

- CCAs can themselves the a source of bugs and should verified as part of unit testing.
- Overuse of `noexcept` can interfere with such unit testing
- **We've considered the alternatives** (you're not as smart as you think you are)!
- Take it away, Timur!

Unit testing

In the original presentation, this was slide 1 for Timur's video.
Please forgive formatting glitches caused by conversion and merging.

Unit testing

Contracts and unit testing complement each other.

Let's unit-test a function with a narrow contract

Let's unit-test a function with a narrow contract:

std::vector::front()

```
// vector_test.cpp
```

```
// vector.h
```

```cpp
// vector_test.cpp
TEST_CASE("vector::front") {
  // TODO
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  // TODO
}
```

```cpp
// vector_test.cpp
TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  // TODO
}
```

⃠ test fails

```cpp
// vector_test.cpp
TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  return data[0];
}
```

✓ test passes

```cpp
// vector_test.cpp
TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  ASSERT(!empty());
  return data[0];
}
```

Test your preconditions!

Why should we test our preconditions?

Why testing preconditions is important

- Preconditions are code – ought to be tested like any other code
- Precondition check ensures users of your API won't introduce bugs!
- Writing test for precondition check ensures:
    - You did not forget to write the precondition check
    - You wrote it correctly
- Ideally, write the test first
  (TDD, "test-driven development")

How do we test our preconditions?

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  // ???
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  ASSERT(!empty());
  return data[0];
}
```

```
#if TEST_ASSERTIONS
  #define ASSERT(expr) if (!expr) throw AssertFail();
#else
  // log & terminate, log & continue, breakpoint, assume, ignore...
#endif
```

```
#if TEST_ASSERTIONS
  #define ASSERT(expr) if (!expr) throw AssertFail();
#else
  // log & terminate, log & continue, breakpoint, assume, ignore...
#endif

#define REQUIRE_ASSERT_FAIL(expr) REQUIRE_THROWS(expr, AssertFail)
```

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() {
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() noexcept {
  ASSERT(!empty());
  return data[0];
}
```

```
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```
// vector.h
template<typename T>
T& vector<T>::front() noexcept {
  ASSERT(!empty()); // std::terminate
  return data[0];
}
```

☹

The Lakos Rule

- We not declare a function with a narrow contract noexcept
  - Even if we know it will never throw (when called in-contract)
  - Instead, we document that it does not throw

```
constexpr const_reference operator[](size_type pos) const;
```

1 *Preconditions*: `pos < size()`.

2 *Returns*: `data_[pos]`.

3 *Throws*: Nothing.

4 [*Note 1*: Unlike `basic_string::operator[]`, `basic_string_view::operator[](size())` has undefined behavior instead of returning `charT()`. — *end note*]

```
constexpr const_reference at(size_type pos) const;
```

5 *Returns*: `data_[pos]`.

6 *Throws*: `out_of_range` if `pos >= size()`.

```
constexpr const_reference front() const;
```

7 *Preconditions*: `!empty()`.

8 *Returns*: `data_[0]`.

9 *Throws*: Nothing.

The Lakos Rule

- [P2861R0] John Lakos: *"The Lakos Rule. Narrow Contracts and noexcept Are Inherently Incompatible"* https://wg21.link/p2861
- [P2831R0] Timur Doumler and Ed Catmur: *"Functions having a narrow contract should not be noexcept"* https://wg21.link/p2831
- [P2837R0] Alisdair Meredith and Harold Bott, Jr.: *"Planning to Revisit the Lakos Rule: The Lakos Rule is Foundational for Contracts"* https://wg21.link/p2837

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() noexcept {
  ASSERT(!empty());
  return data[0];
}
```

Is there any alternative to exception-based testing of preconditions (which requires us to follow the Lakos Rule?)

```
#if TEST_ASSERTIONS
  #define ASSERT(expr) if (!expr) throw AssertFail();
#else
  // log & terminate, log & continue, breakpoint, assume, ignore...
#endif

#define REQUIRE_ASSERT_FAIL(expr) REQUIRE_THROWS(expr, AssertFail)

#if TEST_ASSERTIONS
  #define MY_NOEXCEPT
#else
  #define MY_NOEXCEPT noexcept
#endif
```

```cpp
// vector_test.cpp
TEST_CASE("empty vector::front") {
  vector<int> v;
  REQUIRE_ASSERT_FAIL(v.front());
}

TEST_CASE("vector::front") {
  vector<int> v = {1, 2, 3};
  REQUIRE(v.front() == 1);
}
```

```cpp
// vector.h
template<typename T>
T& vector<T>::front() MY_NOEXCEPT {
  ASSERT(!empty());
  return data[0];
}
```

SUMMARY

The reason libc++ implemented a throwing debug mode handler was for ease of testing. Specifically, I thought that if a debug violation aborted, we could only test one violation per file. This made it impossible to test debug mode. Which throwing behavior we could test more!

However, the throwing approach didn't work either, since there are debug violations underneath noexcept functions. This lead to the introduction of `_NOEXCEPT_DEBUG`, which was only noexcept when debug mode was off.

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.

Testing a precondition check without throwing on failure?

Testing a precondition check without throwing on failure?

- setjmp/longjmp
- child threads
- "stackful coroutines"
- signals
- death tests
  - fork-based
  - clone-based
  - spawn-based

Testing a precondition check without throwing on failure?

- **setjmp/longjmp**
- child threads
- "stackful coroutines"
- signals
- death tests
  - fork-based
  - clone-based
  - spawn-based

17 Language support library [support]

17.13 Other runtime support [support.runtime]

17.13.3 Header <csetjmp> synopsis [csetjmp.syn]

```
namespace std {
  using jmp_buf = see below;
  [[noreturn]] void longjmp(jmp_buf env, int val);
}

#define setjmp(env) see below
```

1 The contents of the header `<csetjmp>` are the same as the C standard library header `<setjmp.h>`.

2 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this document. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any objects with automatic storage duration. A call to `setjmp` or `longjmp` has undefined behavior if invoked in a suspension context of a coroutine ([expr.await]).

SEE ALSO: ISO C 7.13

Testing a precondition check without throwing on failure?

- setjmp/longjmp
- **child threads**
- "stackful coroutines"
- signals
- death tests
  - fork-based
  - clone-based
  - spawn-based

Child threads

- Spawn test in child thread
- assert fail saves info about fail, then locks/freezes thread
- Drawbacks:
  - leaks memory
  - invalidates RAII program logic
  - leaks one thread per test case

Testing a precondition check without throwing on failure?

- setjmp/longjmp
- child threads
- **"stackful coroutines"**
- signals
- death tests
  - fork-based
  - clone-based
  - spawn-based

"stackful coroutines"

- Assert fail yields to cooperative scheduler / event loop
- event loop calls next test
- Drawbacks:
  - leaks memory
  - invalidates RAII program logic
  - unbounded growth of call stack
  - nontrivial to construct
  - no experience in real-world code

Testing a precondition check without throwing on failure?

- setjmp/longjmp
- child threads
- "stackful coroutines"
- **signals**
- death tests
  - fork-based
  - clone-based
  - spawn-based

Signals

- Just a callback mechanism
- Doesn't solve the problem of leaving function

Testing a precondition check without throwing on failure?

- setjmp/longjmp
- child threads
- "stackful coroutines"
- signals
- **death tests**
    - fork-based
    - clone-based
    - spawn-based

Fork-based death tests

- Launch each test in a forked process
- assert fail terminates process
- Drawbacks:
  - requires fast, reliable fork() – UNIX-like systems only!
  - can return only 8 bits of diagnostic data
  - not widely available in unit testing frameworks (only GTest)

Clone-based death tests

- More stable than fork-based
- Drawbacks:
  - Linux only

Spawn-based death tests

- Drawbacks:
  - Requires external framework (DejaGNU, lit, etc.)
  - Requires moving test into other source file
  - Requires building state for each test from scratch
  - Slow

Standard Contracts in C++26...?

```
// C++23
template<typename T>
T& vector<T>::front()
{
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// C++23
template<typename T>
T& vector<T>::front()
{
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// C++26
template<typename T>
T& vector<T>::front()
  [[ pre: !empty() ]]  // P2935R0 syntax
{
  return data[0];
}
```

```cpp
// C++23
template<typename T>
T& vector<T>::front()
{
  ASSERT(!empty());
  return data[0];
}
```

```cpp
// C++26
template<typename T>
T& vector<T>::front()
  pre (!empty())      // P2961R0 syntax
{
  return data[0];
}
```

```
// C++23
#define ASSERT(expr) \
  if (!expr) throw AssertFail();
```

```cpp
// C++23
#define ASSERT(expr) \
  if (!expr) throw AssertFail();
```

```cpp
// C++26
void ::handle_contract_violation
(const std::contract_violation& v) {
  throw AssertFail(v);
}
```

```cpp
// C++23
#define ASSERT(expr) \
  if (!expr) throw AssertFail();
```

```cpp
// C++26
void ::handle_contract_violation
(const std::contract_violation& v) {
  throw AssertFail(v);
}
```

Takeaways

- **Test your preconditions!**
  - Effective to prevent the introduction of bugs across API boundaries due to contract violations
- Best method: **Exception-based testing** (throw on contract violation)
- Requires that functions with preconditions are not noexcept (aka **"Lakos Rule"**)
- Only viable alternative to exception-based testing: **Death tests**. But:
  - Much more complex
  - Much slower
  - Only works on some specific platforms

Agenda 2 – Let's Go Deeper

- Contract checks and testing in depth
- `noexcept` in depth
- Best practices
- A proposed `noexcept` alternative without the pitfalls

In the original presentation, this was slide 19 after Timur's video

Contract Checks and Testing in Depth

Parts of a Defensive Software Framework

CCAs are part of a *defensive software* framework. Such frameworks typically provide three facilities:

1. One or more assert-like macros to check a pre/post condition or invariant.
2. A way to turn checks on and off
   - Usually a macro that controls whether the assert-like macros are active or are noops
3. A user-settable way to control what happens when a violation is detected
   - Usually a *contract violation handler* function installed by the owner of the main program
   - Potentially multiple functions, with different functions called by different macros

A Sample Defensive Framework

```
void violation_handler(const char* file, unsigned line, const char* func,
                       const char* expression);

#ifdef CHECKED_MODE
# define ASSERT(cond) cond ? (void) 0 : \
    violation_handler(__FILE__, __LINE__, __FUNCTION__, #cond)
#else
# define ASSERT(cond) (void) 0
#endif

#define PRECONDITION(cond) ASSERT(cond)
#define POSTCONDITION(cond) ASSERT(cond)
#define INVARIANT(cond) ASSERT(cond)
```

Violation handler (user supplied)

On/off switch

assert-like macro

The Contract-Violation Handler

Possible actions of a user-supplied contract-violation handler (a.k.a. *Oops Handler*) include

- Logging the error and continuing (observe but don't enforce)
- Terminating the program
- Triggering a breakpoint in the debugger
- Executing an endless loop (halt the thread)
- **Throwing an exception**

A Sample Unit-testing Framework

```
struct contract_exception { … };
void test_error(const char* expression);

void violation_handler(const char* file, unsigned line, const char* func,
                       const char* expression) {
  contract_exception(file, line, func, expression);
}

#define TEST_EXPECT(expr) expr ? (void) 0 : test_error(#expr)
#define TEST_EXPECT_PASS(expr) try { (void) (expr); }       \
    catch (const contract_exception&) { test_error("Spurious violation"); }
#define TEST_EXPECT_FAIL(expr) try {                        \
        (void) (expr); test_error("Uncaught violation"); } \
    catch (const contract_exception&) { }
```

Putting it All Together

```
template <class T>
T& vector<T>::operator[](size_type index) {
    PRECONDITION(index <= size());  // BUG!
    return m_data[index];
}

int main() {
    vector<int> v{ 5 };
    int x;
    TEST_EXPECT_PASS(x = v[0]);  // OK
    TEST_EXPECT(5 == x);         // OK
    TEST_EXPECT_FAIL(x = v[1]);  // BUG detected!
    TEST_EXPECT_FAIL(x = v[2]);  // OK
}
```

`noexcept` in depth

Purpose of the `noexcept` specifier

- Added late in the C++11 standardization cycle to allow safe use of move constructors when resizing `vector`-like containers without giving up the *strong exception guarantee*.
- It can often reduce code size, but that was not its purpose.
- It can sometimes make code more self-documenting, but that was not its purpose.
- Unless used with the `noexcept` operator to select a faster code path **the `noexcept` specifier does not make code faster**, despite popular perception.

The Original Intended Use of `noexcept`

```
template <class T, class A>
void vector<T, A>::reallocate(size_type sz) {
  using alloc_traits = allocator_traits<A>;
  pointer new_data = alloc_traits::allocate(m_alloc, sz); // allocate new array

  for (size_type i = 0; i < size(); ++i) {  // Copy or move elements to new array
    if constexpr (noexcept(T(std::move(m_data[i]))))
      alloc_traits::construct(&new_data[i], std::move(m_data[i]));
    else
      try { alloc_traits::construct(&new_data[i], m_data[i]); }
      catch (...) { /* destroy new copies, then rethrow */ }
  }
  …
}
```

Move (efficient)

Copy (safe)

Original elements are unchanged

Summary of Exception Safety Guarantees

- The *basic exception guarantee* – no resources are leaked or corrupted
- The *strong exception guarantee* – all objects are left unchanged
  - Less useful than you might think – only for objects that outlive the innermost `try` block
  - Discouraged for new functions – too much complexity and performance loss for limited benefit
- The `noexcept` operator typically provides *no benefit* for functions having the basic guarantee.

The Basic Exception Guarantee

```
template <class T, class OP>
void xform(std::vector<T>& v, OP operation) {

    for (auto& e : v)
      operation(e);          // inplace modification – might throw

}
```

- What happens if `operation(e)` throws?
  - No resources are leaked, nor memory corrupted
  - **But**, `v` is left partially transformed

Strong Exception Guarantee

```
template <class T, class OP>
void xform(std::vector<T>& v, OP operation) {
    std::vector<T> v2(v);  // copy
    for (auto& e : v2)
      operation(e);         // modify the copy – might throw
    v2.swap(v);             // modify original on success
}
```

expensive and might exhaust memory

- What happens if `operation(e)` throws?
  - No resources are leaked, nor memory corrupted
  - **And**, `v` is left unchanged

Optimized Strong Exception Guarantee

```
template <class T, class OP>
void xform(std::vector<T>& v, OP operation) {
  if constexpr (noexcept(operation(std::declval<T>()))) {
    for (auto& e : v)
      operation(e);          // inplace modification – won't throw
  }
  else {
    std::vector<T> v2(v);   // copy
    for (auto& e : v2)
      operation(e);          // modify the copy – might throw
    v2.swap(v);              // modify original on success
  }
}
```

Optimized path

expensive and might exhaust memory

Normal path

`noexcept` and Versioning

- `negate` has a narrow contract. Version 1 has a `noexcept` interface:

```
void negate(int& val) noexcept;          //  Precondition: val > INT_MIN
```

- For version 2, we want to *widen* the interface, which is typically considered a safe thing to do:

```
void negate(int& val) noexcept(false); //  Throws if val == INT_MIN
```

- Does v2 have the same behavior as v1 for the same inputs?

```
vector<int> v = { … };   //  All values > INT_MIN
xform(v, negate);        //  V1 OK, V2 might exhaust memory!
```

behavior change!

- **NO! V2 is not substitutable for v1.**

`noexcept` is Forever

- A `noexcept` specifier is detectable by the `noexcept` operator
  - It is not a compiler optimization hint like `inline`
  - Adding or removing the specifier changes the *meaning* of the code
- Overuse has a negative effect on unit testing, as we've seen.
- Overuse has a negative effect on future development
  - Once part of a released interface, `noexcept` cannot be removed without breaking backward compatibility.

Best Practices: The Lakos Rule

The Library Working Group (LWG) of the C++ Standards Committee has been following a policy dubbed *The Lakos Rule*, which can be summarized as follows:

- If a function might throw, it must not be marked as `noexcept` (obviously!).
- If a function has a **narrow contract**, it should not be marked as `noexcept`.
- If a constructor is part of a wrapper or proxy, it should be marked as *conditionally* `noexcept`, based on the thing it is wrapping.

Best Practices: Beyond the Lakos Rule

- The `noexcept` specifier is of limited utility
  - It is vital when needed, **but**
  - it is not needed very often – only when dispatching on it using the `noexcept` operator would improve code (i.e., to optimize an algorithm having the strong guarantee)
- Don’t agonize over whether to use the `noexcept` specifier; consider it only
  - For default constructors
  - For move constructors or move-like operations having wide contracts
  - For trivial functions having wide contracts (e.g., returning a class member)
  - Otherwise don't use it

A Proposed Alternative to `noexcept`

A Proposed `[[throws_nothing]]` Attribute

- Standards paper P2946 proposes a `[[throws_nothing]]` attribute to replace the *off-label* uses of `noexcept` without the pitfalls:
  - Can be used to reduce code size
  - Can be used on functions with narrow contracts
  - Can be used to make code self-documenting
- Critically, `[[throws_nothing]]` does not affect the meaning of a correct program:

```
[[throws_nothing]] double myfunc(double val);  // Precondition: x >= 0.
static_assert(! noexcept(myfunc(0.0)));
```

Invisible to the `noexcept` operator

`[[throws_nothing]]` violations

- If a `[[throws_nothing]]` function tries to exit via an exception, the behavior is *implementation defined*.
- Possible behaviors (probably dependent on compiler flags) are
  - Do nothing – let the exception escape — **Allows unit testing CCAs**
  - Terminate, like `noexcept` — **Minimizes code size in production builds**
  - Call the contract violation handler — **Improves failure diagnostics in checked builds. Provides resilience in high-availability systems.**

Conclusion

- Articulate the contracts for your functions, even those parts that cannot be checked in code.
- Check your preconditions when possible.
- Unit-test your precondition checks.
- Minimize your use of the `noexcept` specifier
  - Reserve its use for default constructors and move-like operations
  - Don't use `noexcept` with narrow contracts; follow the Lakos Rule.

Questions?