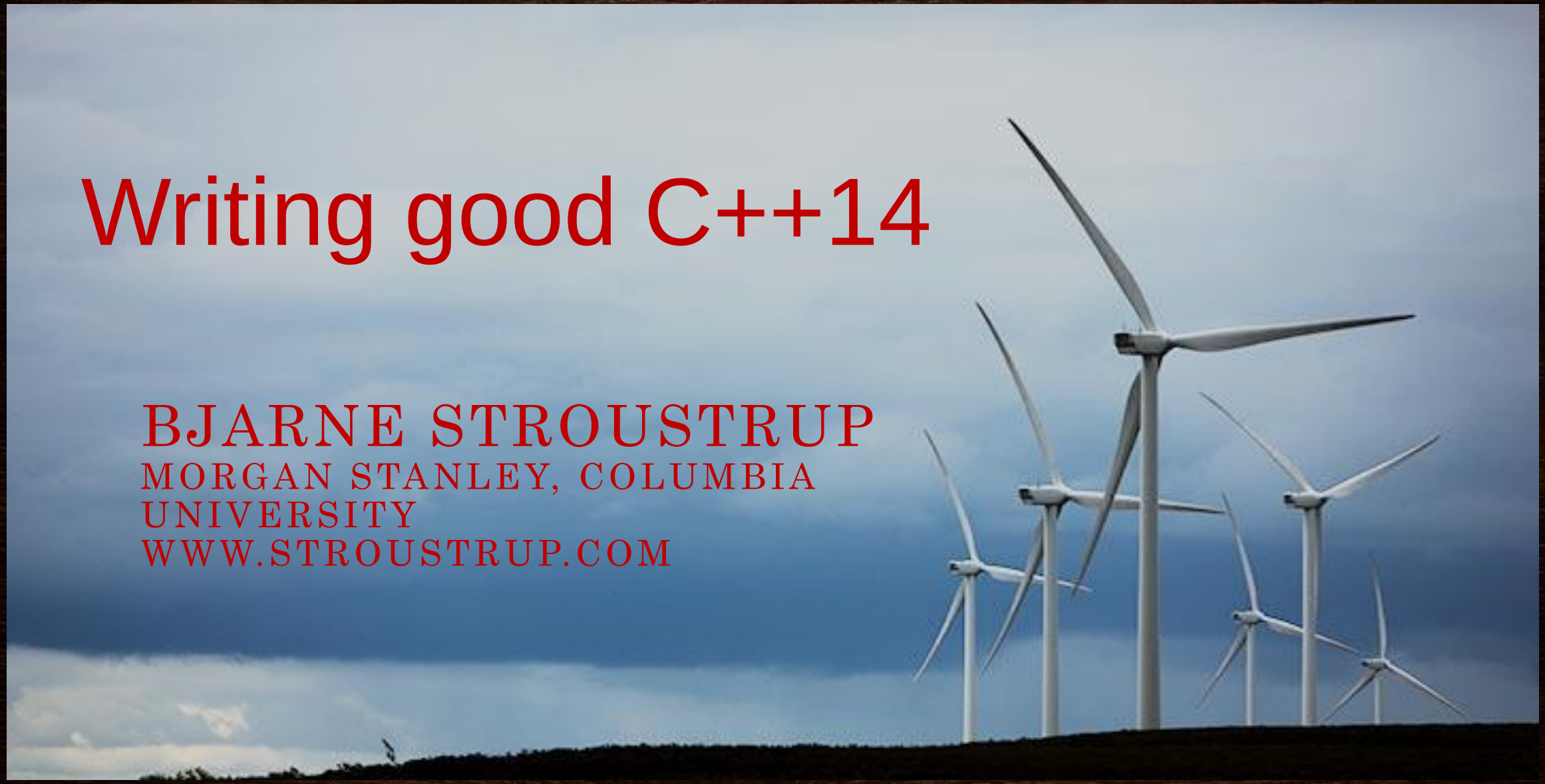
# Contracts for Dependable C++

GABRIEL DOS REIS

# Writing good C++14



BJARNE STROUSTRUP
MORGAN STANLEY, COLUMBIA UNIVERSITY
WWW.STROUSTRUP.COM

# Static Analysis and C++

More Than Lint

Neil MacIntosh
neilmac@microsoft.com

Microsoft

# A Few Good Types

Evolving array_view and string_view
for safer C++ code.

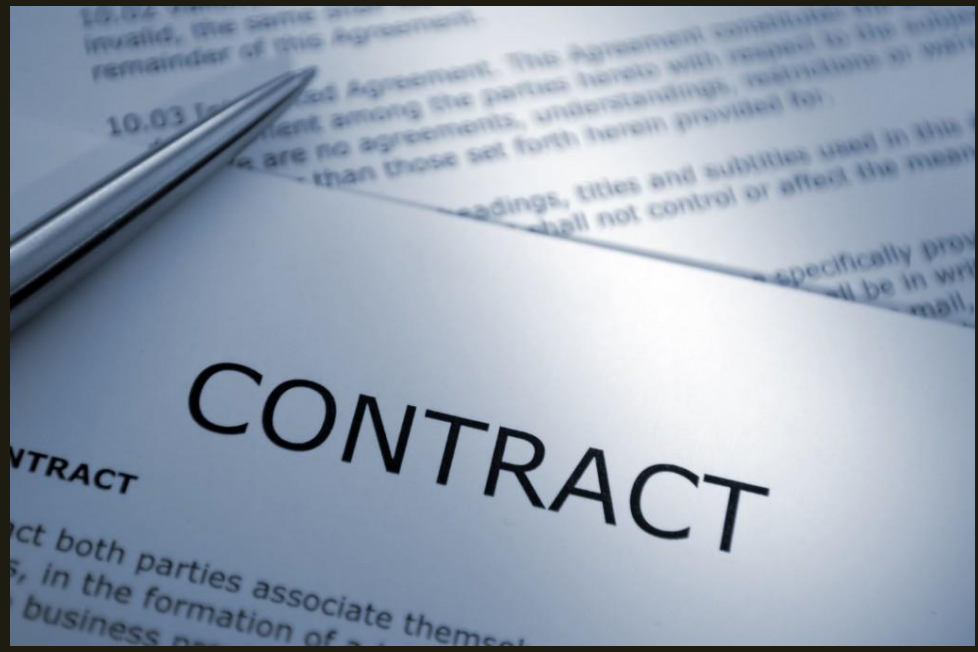
Neil MacIntosh
neilmac@microsoft.com

Microsoft

# From The Core Guidelines (I)

I: Interface

**An interface is a contract between two parts of a program**. Precisely stating what is expected of a supplier of a service and a user of that service is essential.



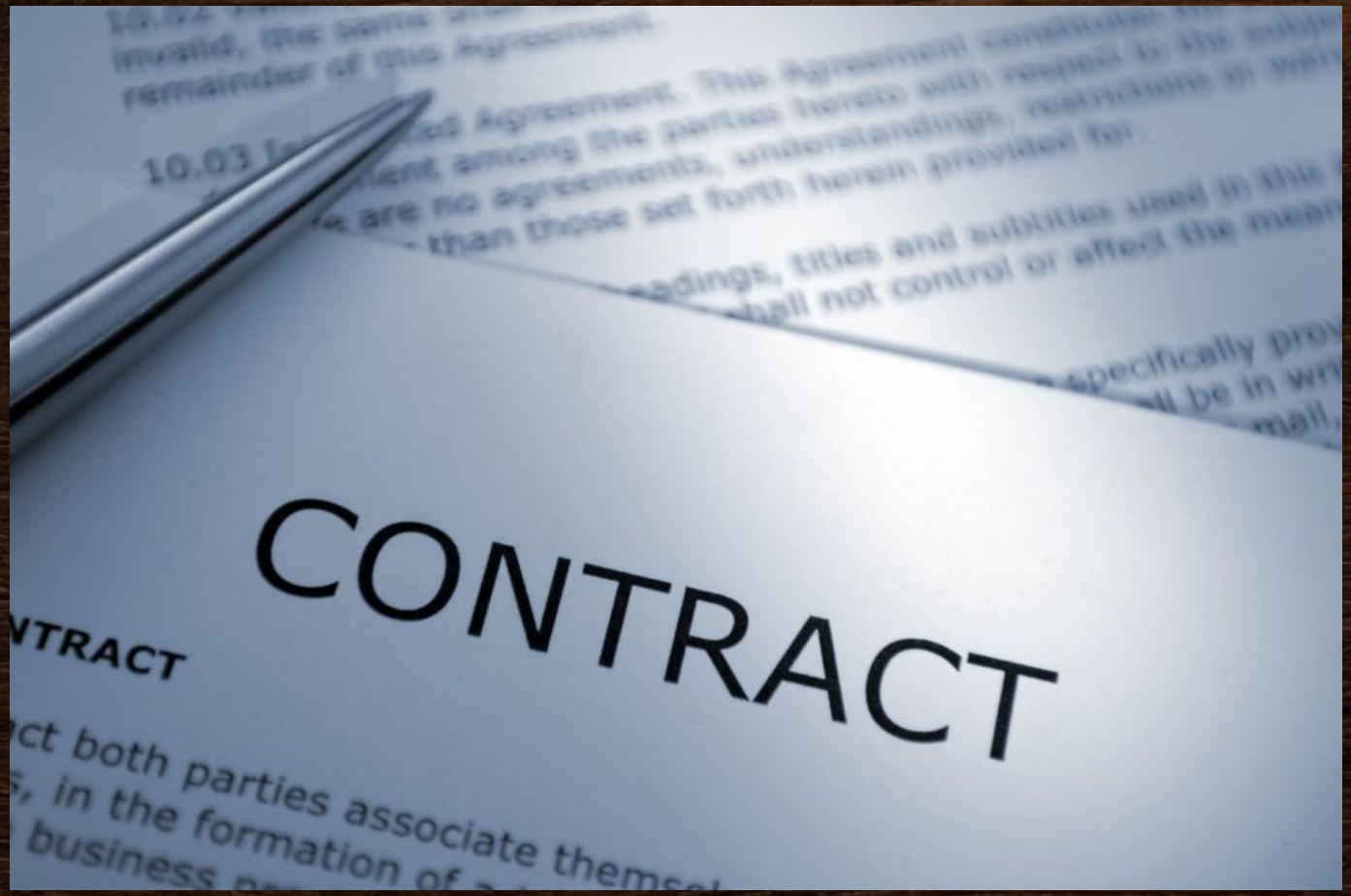

Having good (easy-to-understand, encouraging efficient use, not error-prone, supporting testing, etc.) interfaces is probably the most important single aspect of code organization.

# Acknowledgment

- Lot of work done and lot of papers written on contract for C++
  - In C++0x timeframe
    - Thorsten Ottosen, Lawrence Crawl, Dave Abrahams, …
  - After C++11
    - John Lakos, Nathan Myers, Alisdair Meredith, Alexander Beels, …
    - J.-Daniel Garcia, Lawrence Crawl, Walter Brown, Francesco Logozzo, Manuel Fahndrich, Shuvendu Lahiri, Thomas Ball, GDR, …

- Extensive literature on design by contract, contract programming, etc.

# The Contract Metaphor



# The Contract Metaphor

- Contracts have clauses:

  **A contract clause is a specific provision or section within a written contract.** Each clause in a contract addresses a specific aspect related to the overall subject matter of the agreement. **Contract clauses are aimed at clearly defining the duties, rights and privileges that each party has under the contract terms.**

- LegalMatch

- http://www.legalmatch.com/law-library/article/common-clauses-in-a-contract.html

# Contract Clauses

- Preconditions
- Invariants
- Postconditions

# Contract Clauses

- **Preconditions**
- Invariants
- **Postconditions**

# Precondition

# Precondition

- Interface:
  - The set of requirements that a caller must satisfy before initiating a successful call
    - From consumer's perspective: "What do I have to do to call this function?"

# Precondition

- Interface:
  - The set of requirements that a caller must satisfy before initiating a successful call
    - From consumer's perspective: "**What do I have to do to call this function?**"

- Implementation:
  - The set of assumptions that the callee can make when a call is initiated
    - From the producer's perspective: "**What may I rely on when implementing this function?**"

# From The Core Guidelines (I.5)

I.5: State preconditions (if any)

Reason: Arguments have meaning that may constrain their proper use in the callee

Example: Consider:

```
double sqrt(double x);
```

Here `x` must be nonnegative. The type system cannot (easily and naturally) express that, so we must use other means. For example,

```
double sqrt(double x);  // x must be nonnegative
```

Some preconditions can be expressed as assertions. For example:

```
double sqrt(double x) { Expects(x >= 0); /* … */ }
```

Ideally that `Expects(x >= 0)` should be part of the interface of `sqrt()` but that's not easily done. For now, we place it in the definition (function body).

# From The Core Guidelines (I.5)

I.5: State preconditions (if any) [ *continued* ]

Note: Prefer a formal specification of requirements, such as `Expects(p != nullptr);` If that is infeasible, use English text in comments, such as

```
// the sequence [p, q) is ordered using <
```

Note: Most member functions have as a precondition that some class invariant holds. That invariant is established by a constructor and must be reestablished upon exit by every member function called from outside the class. We don't need to mention it for each member function.

# From The Core Guidelines (I.6)

I.6: Prefer `Expects()` for expressing preconditions

Reason: To make it clear that the condition is a precondition and to enable tool use.

Example:

```
int area(int height, int width)
{
    Expects(height > 0 && width > 0);                  // good
    if (height <= 0 || width <= 0) then my_error();    // obscure
    // …
}
```

Note: Preconditions can be stated in many ways, including comments, `if`-statements, and `assert()`. This can make them hard to distinguish from ordinary code, hard to update, hard to manipulate by tools, and may have the wrong semantics (do you always want to abort in debug mode and check nothing in productions runs?).

# From The Core Guidelines (I.6)

I.6: Prefer `Expects()` for expressing preconditions [ *continued* ]

Note: Preconditions should be part of the interface rather than part of the implementation, but we don't yet have the language facilities to do that.

Note: `Expects()` can also be used to check a condition in the middle of an algorithm.

Enforcement: (Not enforceable) Finding the variety of ways preconditions can be asserted is not feasible. Warning about those that can be easily identified (assert()) has questionable value in the absence of a language facility.

# From The Core Guidelines (I.6)

I.6: Prefer `Expects()` for expressing preconditions [ *continued* ]

Note: Preconditions should be part of the interface rather than part of the implementation, but we don't yet have the language facilities to do that.

Note: `Expects()` can also be used to check a condition in the middle of an algorithm.

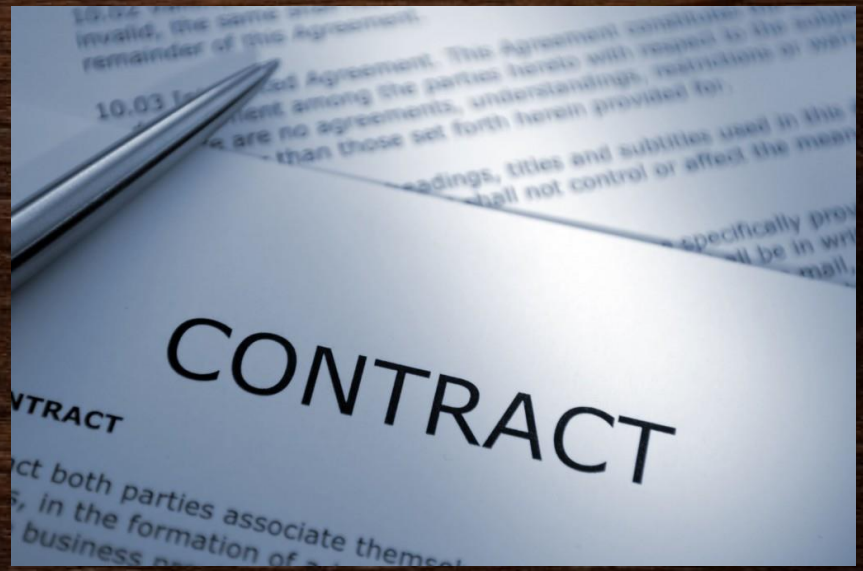
Enforcement: (Not enforceable) Finding the variety of ways preconditions can be asserted is not feasible. Warning about those that can be easily identified (assert()) has questionable value in the absence of a language facility.

# Postconditions

# Postconditions

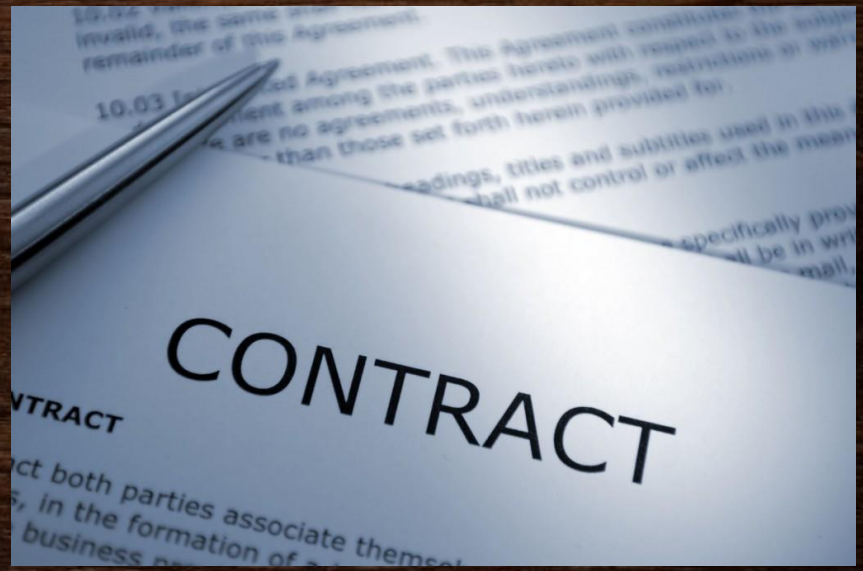
- Interface
  - The set of assumptions that a caller can make when a call completes
    - From consumer's perspective: "What may I rely on when this function returns?"

- Implementation
  - The set of requirements that a callee must fulfill before transferring control back to the caller
    - From implementer's perspective: "What do I need to do before returning?"


CONTRACT

# Breach of Contract

- Program crashes
  - Lucky
- Bad press (lucky)
  - Security leak
- Economic disaster
- Deaths
  - Crashes in critical moments
- Undefined Behavior
  - Unrestricted behavior



# In Search of Lightweight Abstractions

# In Search of Lightweight Abstractions

- Direct language support
- Support safer C++
  - Tooling
- Enable new sound programming styles
- Contain “undefined behavior”
  - Dependable systems
- Stay true to C++
  - Don’t pay for what you don’t use

# Design Principles

- Avoid making assert() a wonderful, preferred alternative
- Avoid major modifications to the type system
  - For correct program instances, deleting contracts should be unobservable
- Interoperate with, and leverage existing abstraction mechanisms

# Syntax: Strawman

- Attribute-based syntax:
  - Precondition: `[[expects: condition]]`
  - Postcondition: `[[ensures: condition]]`
- **Note**: Erasing contracts from a correct program instance should not change the observable behavior
  - Attributes fit well

# Example

```
template<typename T>
struct ArrayView {
    // …
    int size() const;
    T* data() const { return elem; }
    T& operator[](int i) [[expects: i >= 0 and i < size()]]
    {
        return data()[i];
    }

private:
    T* elem;
    int span;
};

int main() {
    Vector<int> vec { 304, 5, 6565, 234, 545 };
    ArrayView<int> view { vec,  4 };
    view[2] += 42;                                  // OK: in contract
    view[9] = 7;                                    // Out of contact
}
```

# Example

```
template<typename T>
struct ArrayView {
    ArrayView(const Vector<int>& v, int n)
        [[ensures: data() == v.data() and size() == n]];
    int size() const { return span; }
    T* data() const { return elem; }
    T& operator[](int i) [[expects: i >= 0 and i < size()]]
    {
        return data()[i];
    }

private:
    T* elem;
    int span;
};

int main() {
    Vector<int> vec { 304, 5, 6565, 234, 545 };
    ArrayView<int> view { vec,  4 };
    for (auto& x : view)
        x = sqrt(x) + 3;
}
```

# Semantics

- Precondition `[[expects: condition]]`
  1. Arguments are evaluated
  2. condition is evaluated
  3. Out-of-contract counter-measure enacted if contract violated
  4. First statement of the user-authored function body executed

# Semantics

- Precondition `[[expects: condition]]`
  1. Arguments are evaluated
  2. `condition` is evaluated
  3. Out-of-contract counter-measure deployed if contract violated
  4. First statement of the user-authored function body executed
- Postcondition: `[[ensures: condition]]`
  1. User-authored function body executed (except return)
     - Return expression, if any, evaluated
  2. condition is evaluated
  3. Out-of-contract counter-measure deployed if contract violated
  4. Control transferred to caller.

# Contract Validation: When and Where?

- Is a contract condition always evaluated?

# Contract Validation: When and Where?

- Is a contract condition always evaluated?
  - No.
    - Depending on circumstances, it may make sense to turn some checks off.

- So, how do I turn a check off in the source code?
  - You don't get to turn off check in the source code
    - Maybe on translation unit bases, per-module basis, per-component basis

# Contract Validation: When and Where?

- Is a contract condition always evaluated?
  - No.
    - Depending on circumstances, it may make sense to turn some checks off.
- So, how do I turn a check off in the source code?
  - You don't get to turn off checks in the source code
    - Maybe on translation unit bases, per-module basis, per-component basis
- How many check levels can I request?
  - A few
    - on, off, and in-between
- Can I mix translation units with different checking levels?
  - This is a novel concepts for standard ISO C++
    - Not entirely in practice

# Implementation

# Contracts for Dependable C++