





## AGENDA

- What is a function contract?
- Why a narrow contract (one with preconditions) might be a good choice?
- How to design a good function contract?
- How to check that contracts are not violated?
- How to communicate a contract to our clients?
- What is required of a contract-checking facility at scale?

# WHAT IS A FUNCTION CONTRACT?

## WHAT IS A ~~FUNCTION~~ CONTRACT?

*A contract is an agreement between two parties that creates an obligation to perform (or not perform) a particular duty.*

**<https://www.utsa.edu/bco/resources/contract-law-101.html>**

*A contract is an agreement, either written or spoken, between two or more parties that creates a legal obligation.*

**<https://ironcladapp.com/journal/contracts/what-is-a-contract/>**

*A contract is an agreement between parties, creating mutual obligations that are enforceable by law.*

**<https://www.law.cornell.edu/wex/contract>**

## EXAMPLE CONTRACT

The contractor will provide manpower (labor), rough materials, small tools, equipment and schedule subcontractors for the work as outlined in this contract. The work will be performed following standard building practices. The contractor will supply enough manpower to have this job substantially completed in 8 weeks. This proposed schedule does not account for time in hold for inspections or other unexpected circumstances (i.e.: weather storms, etc.)

Owner's responsibilities and Progress Requisition:

1. Payment: The failure of the Owner to make proper payments to Contractor when due shall, in addition to all other rights, constitute a material breach of contract and shall entitle Contractor, at its discretion, to suspend all work. Owner agrees to pay interest at a rate of 1.5 percent per month retroactive to the completion date on any amount not paid when due.
2. Bi-weekly requisition payments will be paid assuming reasonable bi-weekly progress. Requisition payments can be delayed or accelerated as needed upon agreement by the parties.

## WHAT DO CONTRACTS HAVE TO DO WITH SOFTWARE?

- The two parties
  - The function author
  - The client code
- The obligations
  - (Author) Function promises to perform a task
  - (Client) Caller promises to respect constraints on inputs (if any)
- Enforceable by law?
  - Not really...
  - ... but we'll talk about enforcement

## DISCLAIMER

- All examples will use C++
- Most (if not all) ideas regarding contracts are applicable to any programming language
- Code is inspired by
  - decades of using contracts at Bloomberg
  - "C++20" contracts that never happened
  - current work towards C++26 contracts (SG21)



## ELEMENTS OF A FUNCTION CONTRACT

- Preconditions – constraints that the client must satisfy
  - With respect to input arguments and object or program state
  - ... otherwise, the behavior is undefined
- Essential behavior – what a function promises to do given valid input
  - Postconditions
    - Return value(s)
    - Changes in the object or program state
  - Behavioral guarantees
    - Algorithmic complexity
    - Thread safety
    - ...

## WHY HAVE PRECONDITIONS AT ALL?

- Some functions don't have preconditions
  - Naturally wide contracts
    - `std::vector::push_back()`
    - `std::vector::size()`
  - Other functions are not quite that easy
    - `std::vector::front()`
    - `std::vector::operator[](index)`
    - `std::ranges::sort(range, comparator)`

## REASONS TO HAVE PRECONDITIONS

- Defining behavior for all inputs might be inefficient or even infeasible
- Having appropriately selected preconditions improves
  - Reliability
  - Maintainability
  - Extensibility

## FEASIBILITY

- Defining behavior for all inputs might be impossible no matter how hard we try
  - `basic_string_view::basic_string_view(const CharT* s, size_type count);`
    - Precondition: `[s, s+count)` is a valid range
  - `std::ranges::sort(rng, comp)`
    - Precondition: `comp` must impose a strict weak order
  - `std::mutex::lock()`
    - Precondition: lock must not be owned by the thread that calls `lock()`

## EFFICIENCY

- Defining behavior for all inputs might be too expensive
  - `std::ranges::merge(rng1, rng2, result) ← O[N]`
    - Precondition: `rng1` and `rng2` must be sorted with respect to `operator<` ←  $O[N]$
  - `std::ranges::lower_bound(rng, value) ← O[\log(N)]`
    - Precondition: `rng` must be partitioned with respect to `operator<` ←  $O[N]$
- Performance penalty is present on every call

# RELIABILITY

- "Handling" nonsense inputs masks defects
  - Some hypothetical BAD examples
    - `std::vector::pop_back()` does nothing when vector is empty
    - `std::string_view::operator[](index)` returns the last character if index is bigger than length
    - `std::optional::operator*()` returns a reference to a static default-constructed value when empty
  - If there's a defect in the caller such behavior might...
    - ... "work" in some situations giving false sense of security
    - ... manifest as an error/crash/etc. at code location far away, complicating debugging

## MAINTAINABILITY

- "Handling" nonsense input complicates implementation
- More error paths complicate client code

```
1 double min_double(const double *begin, const double *end)
2 {
3     const double *cur = begin;
4     const double *min = begin;
5     while (++cur < end) {
6         min = *min < *cur ? min : cur;
7     }
8     return *min;
9 }
```

## MAINTAINABILITY

- "Handling" nonsense input complicates implementation
- More error paths complicates client code

```
1  double min_double(const double *begin, const double *end)
2  {
3      if (nullptr == begin)          throw std::invalid_argument{"null begin"};
4      if (nullptr == end)           throw std::invalid_argument{"null end"};
5      if (std::less<>{}(end, begin)) throw std::invalid_argument{"bad range"};
6      if (begin == end)             throw std::invalid_argument{"empty range"};
7
8      const double *cur = begin;
9      const double *min = begin;
10     while (++cur < end) {
11         min = *min < *cur ? min : cur;
12     }
13     return *min;
14 }
```



## MAINTAINABILITY

- "Handling" nonsense input complicates implementation
- More error paths complicates client code

```
1  double min_double(const double *begin, const double *end)
2  {
3      if (nullptr == begin)          throw std::invalid_argument{"null begin"};
4      if (nullptr == end)           throw std::invalid_argument{"null end"};
5      if (std::less<>{}(end, begin)) throw std::invalid_argument{"bad range"};
6      if (begin == end)             throw std::invalid_argument{"empty range"};
7
8      const double *cur = begin;
9      const double *min = begin;
10     while (++cur < end) {
11         if (std::isnan(*cur))      throw std::domain_error{"unexpected NaN"};
12
13         min = *min < *cur ? min : cur;
14     }
15     return *min;
16 }
```

## MAINTAINABILITY

- "Handling" nonsense input complicates implementation
- More error paths complicates client code

```
1  double min_double(const double *begin, const double *end)
2  {
3      if (nullptr == begin)          throw std::invalid_argument{"null begin"};
4      if (nullptr == end)           throw std::invalid_argument{"null end"};
5      if (std::less<>{}(end, begin)) throw std::invalid_argument{"bad range"};
6      if (begin == end)             throw std::invalid_argument{"empty range"};
7
8      const double *cur = begin;
9      const double *min = begin;
10     while (++cur < end) {
11         if (std::isnan(*cur))      throw std::domain_error{"unexpected NaN"};
12
13         min = *min < *cur ? min : cur;
14     }
15
16     if (min < begin || end <= min) throw std::logic_error{"algorithm failed"};
17     if (cur != end)               throw std::logic_error{"algorithm failed"};
18     return *min;
19 }
```

## EXTENSIBILITY

- We can't easily change defined behavior
  - Breaks clients relying on that behavior
  - We can define the behavior that was previously undefined
  - Any correct client won't feel the difference\*
- Example: If we were to make `my_vector::operator[]` abort on out-of-range, no existing correct program would break

```
1 const_reference operator[] (size_type position) const
2 {
3     return d_dataBegin_p[position];
4 }
```

\*Hyrum's Law: With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody

## EXTENSIBILITY

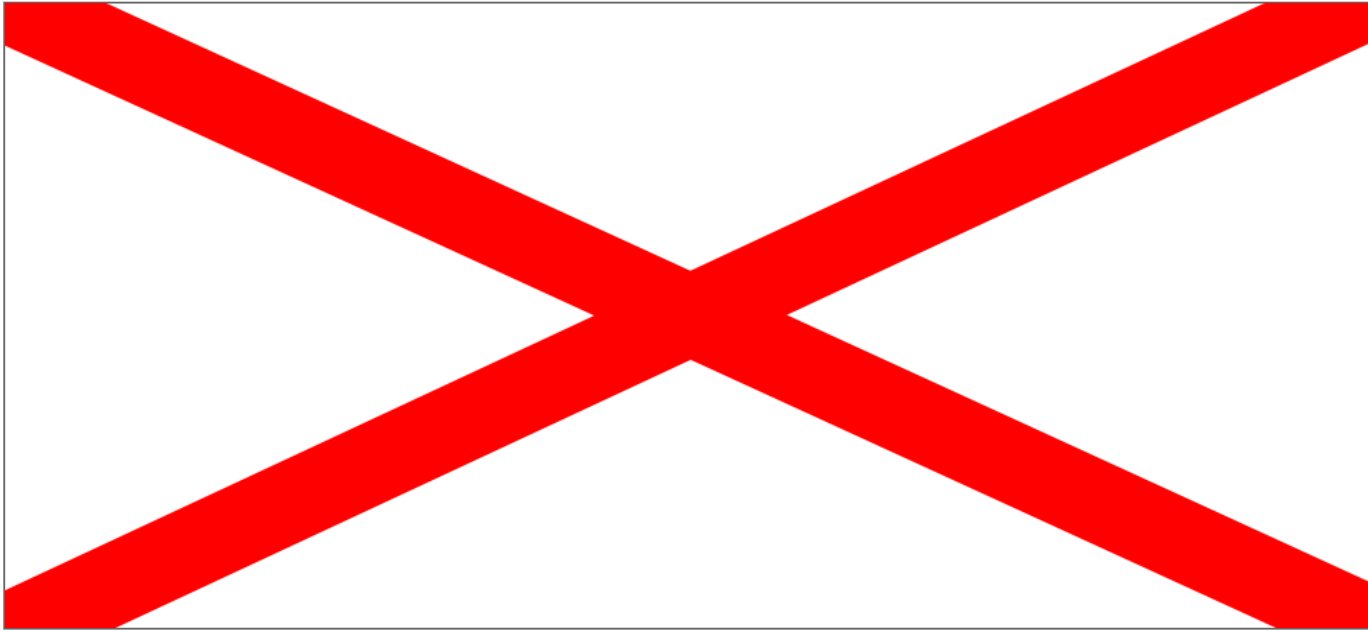
- We can't easily change defined behavior
  - Breaks clients relying on that behavior
  - We can define the behavior that was previously undefined
  - Any correct client won't feel the difference\*
- Example: If we were to make `my_vector::operator[]` abort on out-of-range, no existing correct program would break

```
1 const_reference operator[] (size_type position) const
2 {
3     if (position >= size()) { std::abort(); }
4     return d_dataBegin_p[position];
5 }
```

# **GOOD CONTRACT DESIGN**

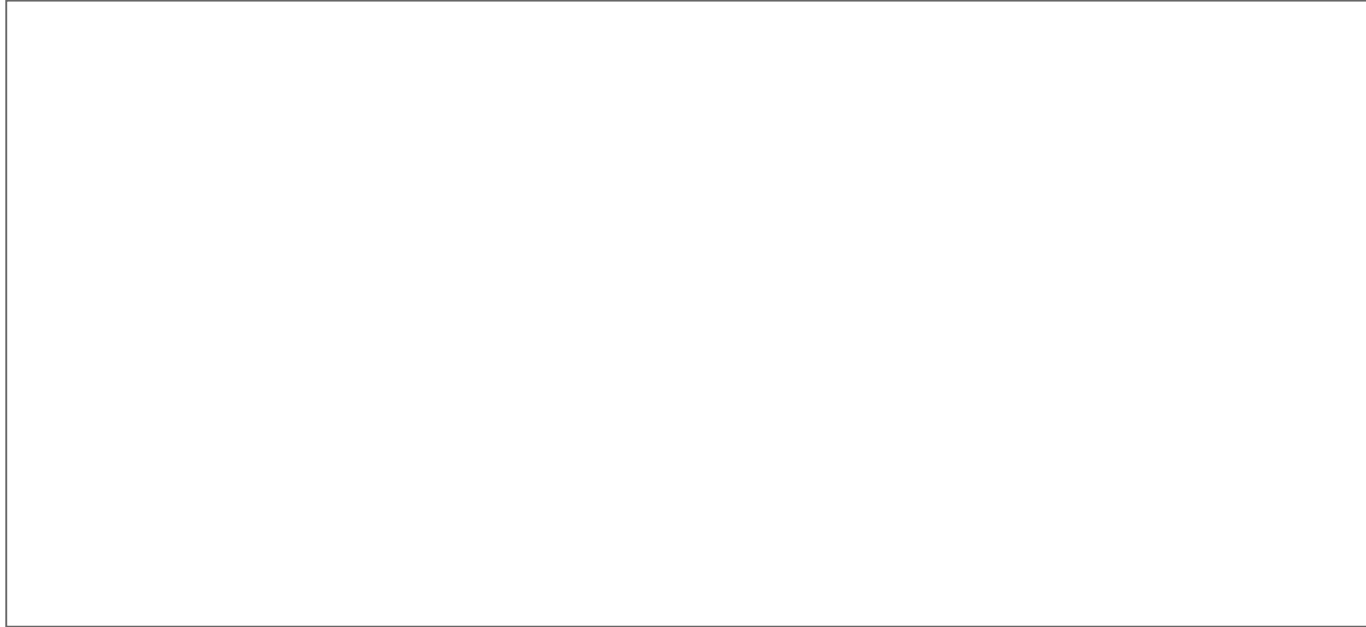
## ORDER OF OPERATIONS

- In what order do we do things?



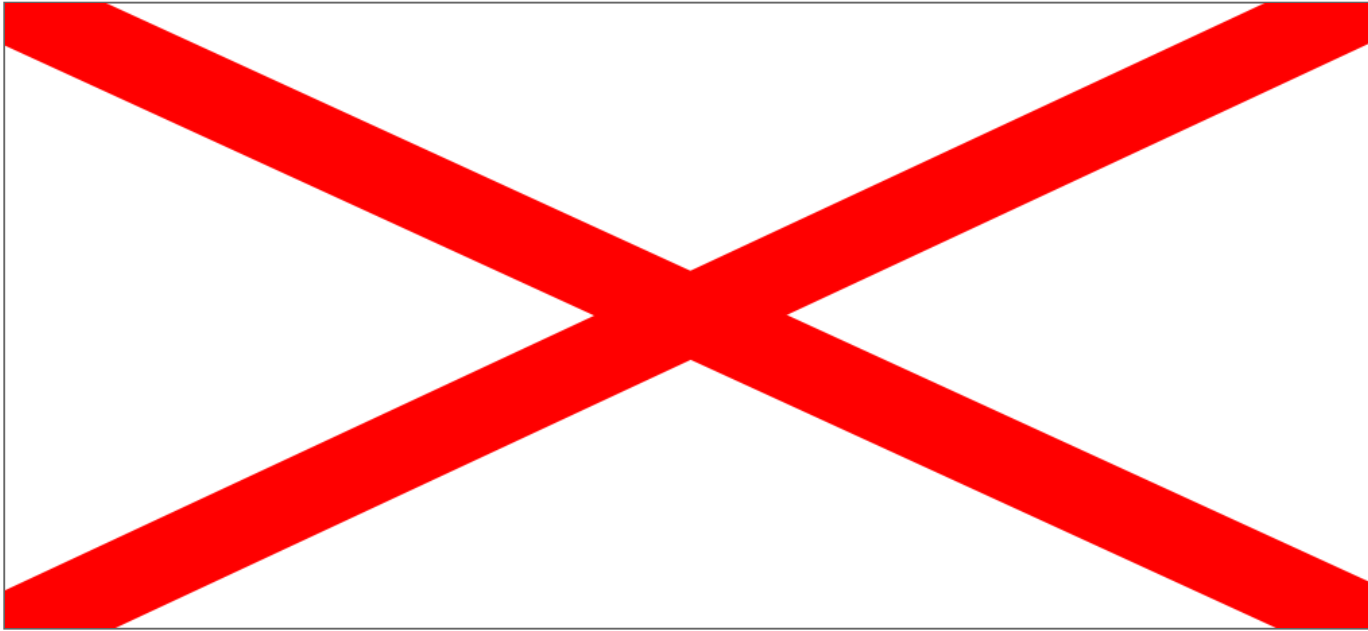
## ORDER OF OPERATIONS

- In what order do we do things?

A large, empty rectangular box with a thin black border, intended for the student to write their answer to the question.

## ORDER OF OPERATIONS

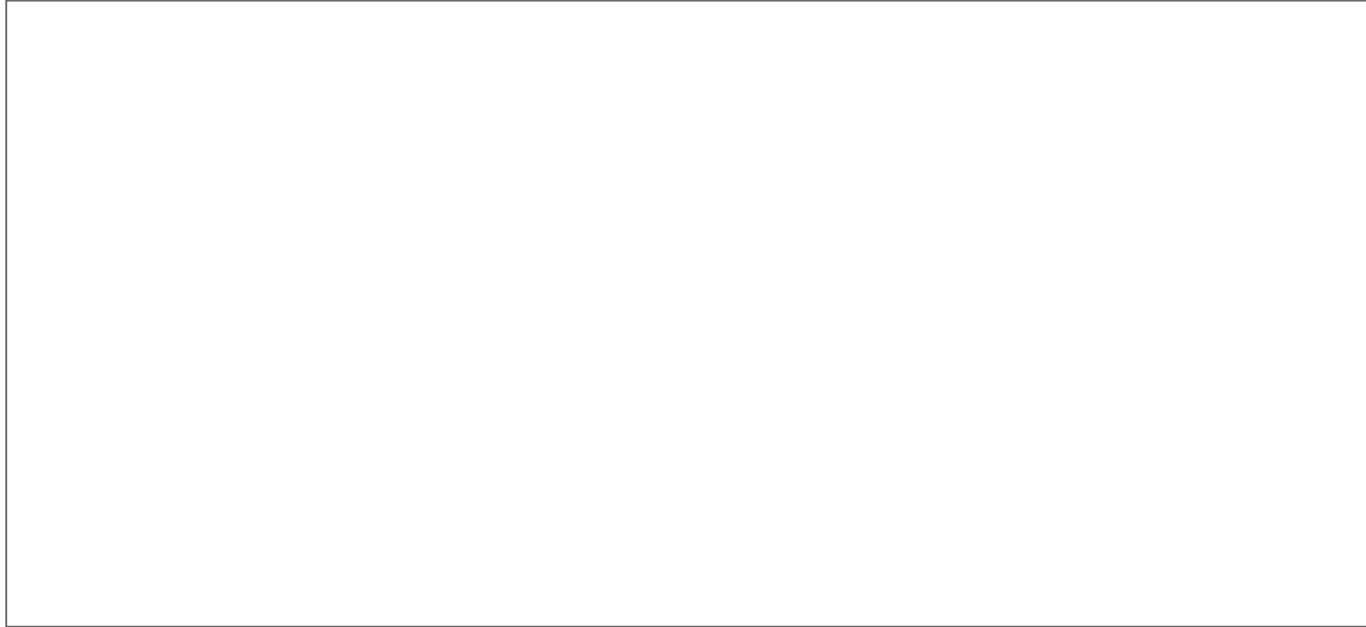
- In what order do we do things?





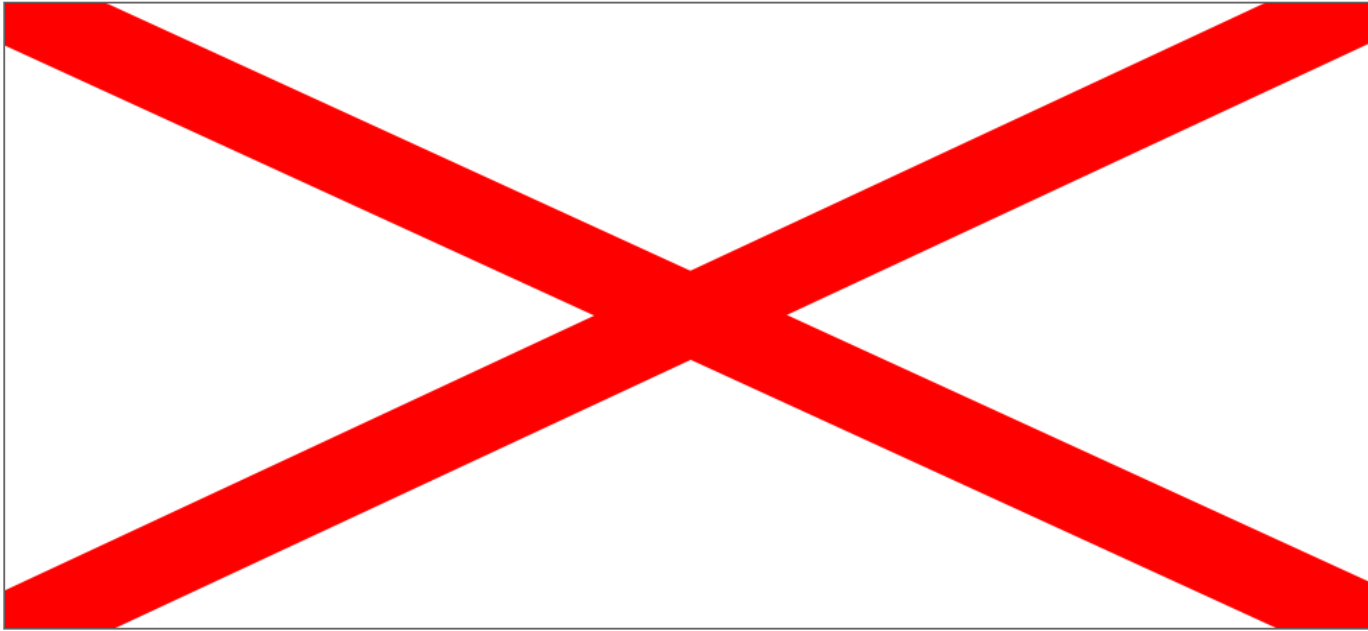
## ORDER OF OPERATIONS

- In what order do we do things?

A large, empty rectangular box with a thin black border, intended for the student to write their answer to the question.

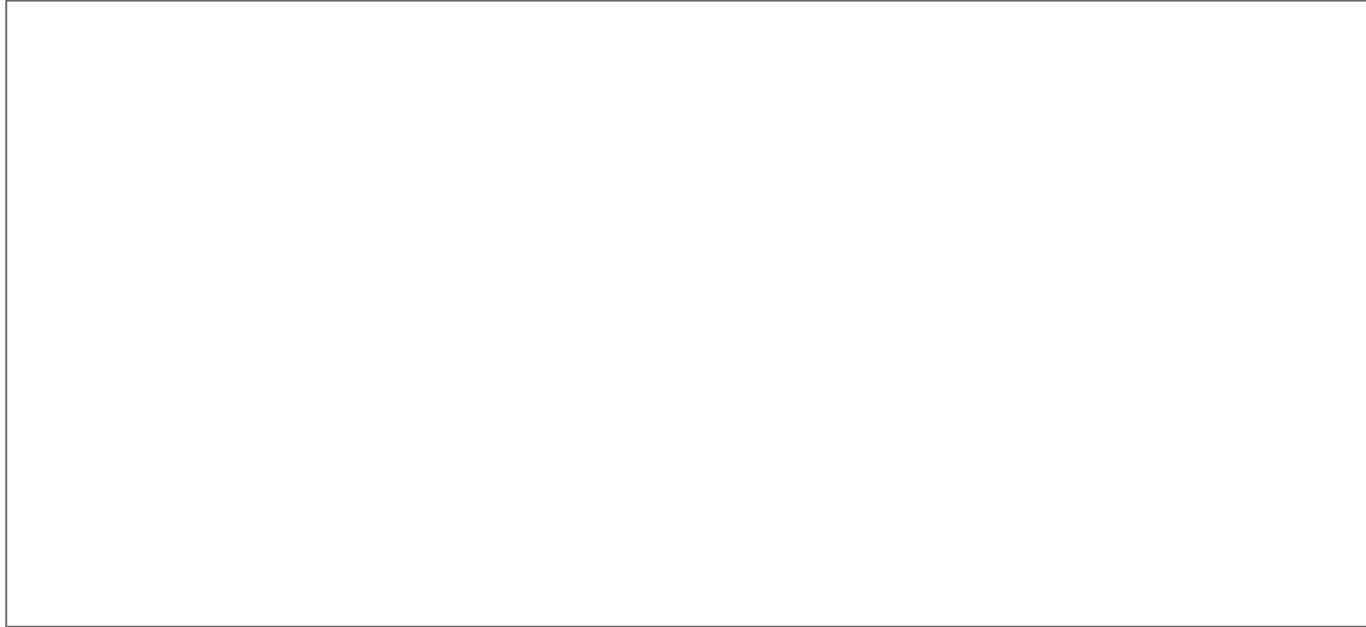
## ORDER OF OPERATIONS

- In what order do we do things?



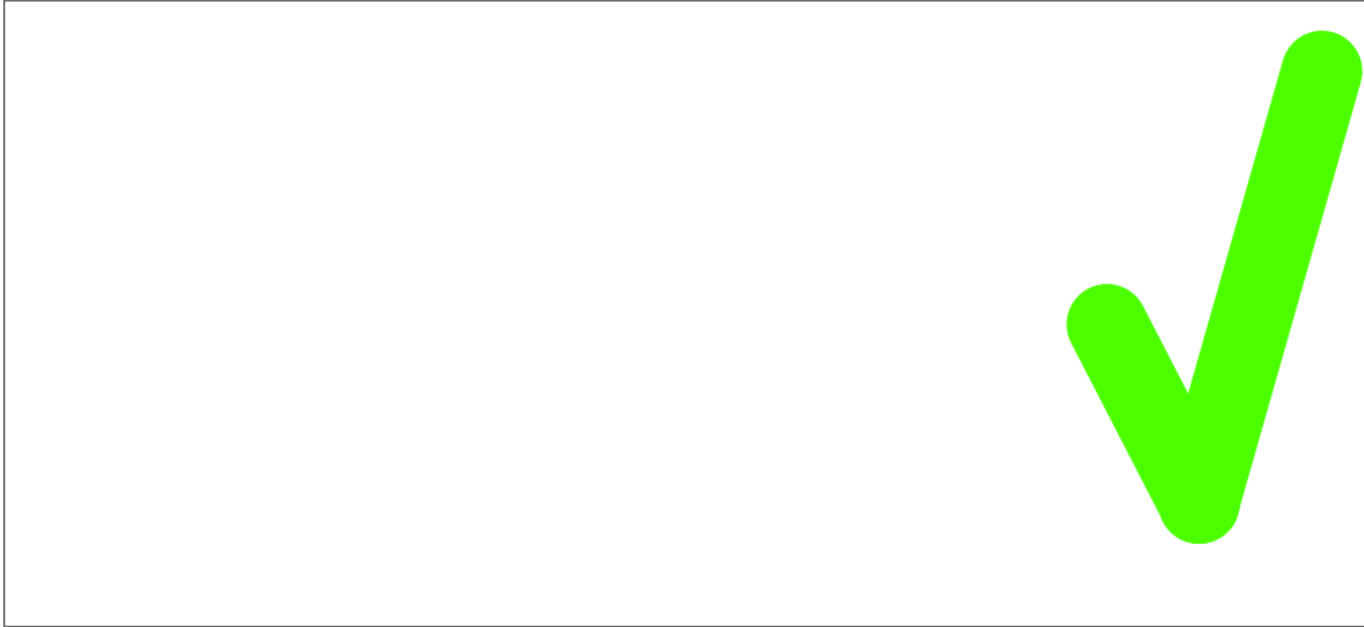
## ORDER OF OPERATIONS

- In what order do we do things?

A large, empty rectangular box with a thin black border, intended for a student to write their answer to the question above.

## ORDER OF OPERATIONS

- In what order do we do things?



## **ASPECTS OF A GOOD CONTRACT**

- Minimal but complete
  - Does the job that it needs to do
  - Does not do what it doesn't need to do
  
- Suitable for a wide variety of clients
  - Does not impose an unnecessary penalty

## NATURALLY WIDE CONTRACTS

- `std::vector::push_back()`
- `std::vector::size()`
- `std::recursive_mutex::lock()`
  - What if the number of levels of ownerships is exceeded?
  - Would such lack of system resources necessarily indicate misuse?
  - Similar to a failure to allocate memory
    - Don't be hostile to your clients; keep the contract wide!
    - Standard chose to throw `std::system_error` in such a scenario

## NARROW BUT NOT TOO NARROW

```
void std::string_view::remove_prefix(size_type count);  
    // Move the beginning of this view by the  
    // specified `count` characters.
```

- Should this function handle `count == 0`? Yes!
- Should this function handle `count == size()`? Yes!
- Same implementation allows us to naturally handle these cases
- What should happen if `count > size()`?
  - Undefined behavior!

## WIDE CONTRACT FACILITIES

- `void HttpHeaders::addField(name, value);`
- HTTP/2 places complex constraints on name and value
- Should it return a status or indicate problems in name and value? No!
  - Would have to check in all build modes
  - Potential errors would need to be handled by the clients
- Instead, provide validation functions and wide-contract facilities:
  - `static bool HttpHeaders::isValidNameValue(name, value);`
  - `int HttpHeaders::addFieldIfValid(name, value);`



```
1 int HttpHeader::addFieldIfValid(const std::string_view& name,  
2                               const std::string_view& value)  
3 {  
4     if (isValidNameValue(name, value)) {  
5         addField(name, value);  
6         return 0;  
7     }  
8     return -1;  
9 }
```

# CONTRACT CHECKING

## **BUGS HAPPEN**

- What to do when a contract is violated?
- Per our contract, behavior is undefined
  - We can do whatever we want!
  - Call law enforcement?
  - Sue for damages?
  - Do something to help our caller understand and fix the bug!

## WHAT ARE OUR OPTIONS?

```
void HttpHeaders::addField(const std::string_view& name,  
                           const std::string_view& value);  
    // [...] The behavior is undefined unless 'isValidNameValue(name, value)'.
```

- Do nothing?
- Check whether precondition was violated and...
  - ...try to fix the incorrect characters?
  - ...throw an exception?
  - ...print a useful message?
  - ...abort the program?
  - ...print a useful message and abort?

## IS CHECKING MANDATORY?

- Should we *always* check?
  - Do we have to check *everything*?
  - **No!**
- 
- Undefined behavior is just that, *undefined*!

## EXAMPLE: C assert

```
1 #include <cassert>
2
3 void HttpHeaders::addField(const std::string_view& name,
4                             const std::string_view& value)
5 {
6     assert(isValidNameValue(name, value));
7     // [...] Implementation
8 }
```

## WHAT DOES AN `assert` GET US?

- In a checked build (`NDEBUG` is not defined)
  - A bug (contract violation) is detected
  - Early
  - (Often) close to the source

```
output.s: /app/example.cpp:12:  
void addField(const std::string_view &, const std::string_view &):  
Assertion `isValidNameValue(name, value)' failed.
```

- In an unchecked build (`NDEBUG` is defined)
  - Run time efficiency – the check is elided

## AREN'T WE IN UB LAND?

- When a contract is violated, we've often not yet entered the Language UB land
  - Instead, we have *Library* UB

```
1 size_t strlen(const char *str)
2 {                               // ← Library (but not Language UB)
3                                 //   might have occurred here
4     size_t count = 0;
5     while (str[count])         // ← Language UB might be here
6         ++count;
7     return count;
8 }
```



## CONTRACT CHECKING

- Contract checks are *redundant* code aiming to detect misuse by the function caller
- Removing some or all checks from a *correct* program should not affect its *essential* behavior

## SIDE EFFECTS IN PREDICATES

- The behavior is undefined if inserting value into `setOfIntegers` fails
- Is this a good way to check it?

```
assert(setOfIntegers.insert(value).second);
```

BAD IDEA!

## SIDE EFFECTS IN PREDICATES

- The behavior is undefined if inserting value into `setOfIntegers` fails
- Is this a good way to check it?

```
auto [_, inserted] = setOfIntegers.insert(value);  
assert(inserted);
```

OK!

## SIDE EFFECTS IN PREDICATES

```
1  class EncryptedStore {
2      std::map<int, int> d_map;
3
4      public:
5
6      void insertValue(int index, int value)
7          // The behavior is undefined if the value at 'index' has been
8          // previously corrupted.
9      {
10         assert(!isCorrupted(d_map[index])); // ← Is this a good check?
11         // [...] Implementation
12     }
13 };
```

**BAD IDEA!**

## BENIGN SIDE EFFECTS IN CONTRACT CHECKS

```
1  class HttpHeader {
2      std::map<std::string, std::string> d_fields;
3
4      public:
5          bool contains(const std::string_view& name) const
6          {
7              LOG_TRACE << "Checking whether '" << name << "' is present among "
8                  << d_fields.size() << " fields.";
9
10             return d_fields.find(std::string(name)) != d_fields.end();
11         }
12
13         void addField(const std::string_view& name, const std::string_view& value)
14             // [...] The behavior is undefined if a field 'name' has already been
15             // inserted.
16         {
17             assert(!contains(name)); // ← Is this a good check?
18             // [...] Implementation
19         }
20     };
```

Some side effects might be OK, depending on your application.

## CHECKS SHOULD NOT AFFECT CONTROL FLOW

```
1  std::optional<int> halfValueOrNull(const std::vector<int>& values, size_t index)
2      // Return half of the value in the specified 'evenValues' at the specified
3      // 'index' if 'index' is in range and 'nullopt' otherwise. The behavior is
4      // undefined unless the value at 'index' is even.
5  {
6      try {
7          assert(values.at(index) % 2 == 0);
8          return values[index] / 2;
9      }
10     catch (const std::out_of_range&)
11     {
12         return std::nullopt;
13     }
14 }
```

BAD IDEA!

## CONTRACT CHECKS ARE NOT FOR INPUT VALIDATION

- Input is data coming from untrusted sources
  - User input via UI or command line
  - Data read from a file
  - Data received over the wire
- Using contract checks for input validation is a BAD idea!
  - `assert(argc == 2); // BAD IDEA!`
- For more, see **[Avoid Misuse of Contracts, CppCon 2019](#)**

## CONTRACT CHECKS DO NOT REPLACE UNIT TESTING

- Contract checks *complement* unit testing
  - Contract checks help find defects while unit testing
- Unit tests verify postconditions
- Asserting invariants in destructors help unit tests to verify them
  - A good unit test would put an object in every possible state before destruction



## SUMMARY: FUNCTION CONTRACT BASICS

- When creating a function, start with defining its contract
- The contract should be minimal but complete
  - Leave behavior undefined for bad inputs (preconditions)
  - Having preconditions allows for faster, simpler, more robust, and more flexible code
- Use contract checking to defend against caller misuse
  - Help your clients catch defects early and precisely
- Do not misuse contract checks
  - Do not put essential side effects in contract checks
  - Do not use contract checks as a control flow mechanism
  - Do not use contract checks for input validation
- Contract checks complement unit testing

## **CONTRACT RENDERING (FOR HUMANS)**

## WHY IS CONTRACT DOCUMENTATION IMPORTANT?

- Why not just "self-documenting code"?

```
1 Quaternion(float w, float x, float y, float z)
2 : d_w(w)
3 , d_x(x)
4 , d_y(y)
5 , d_z(z)
6 {}
```

- Documenting your contract first encourages thinking through the problem
- Any behavior becomes essential behavior
- Not all preconditions can be checked

## **BASIC CONTRACT DOCUMENTATION GUIDELINES**

- All aspects of the contract must be documented
- Specific style doesn't matter much
- Consistency is *key!*

## **ANOTHER DISCLAIMER**

- I will employ the style used by my team (BDE)
- I'll make a direct comparison with a different style toward the end of the section

## **ASPECTS OF A FUNCTION CONTRACT**

- What the function does
- What it returns
- Essential behavior
- Preconditions
- Additional notes

## WHAT THE FUNCTION DOES

- A single imperative sentence describing the primary action(s) the function performs
  - Difficulty fitting the purpose in a single sentence indicates a design issue
- Call out all (non-optional) arguments by their name, explaining how they are used/modified by the function
  - Sentence flow breakage indicates a naming or design issue

## WHAT THE FUNCTION DOES: EXAMPLE

```
1 iterator basic_string::erase(const_iterator first, const_iterator last);  
2     // Erase from this string a substring defined by the specified pair of  
3     // 'first' and 'last' iterators within this string. [...]
```



## WHAT THE FUNCTION RETURNS

- Single imperative sentence describing the return value of the function

```
1 iterator basic_string::erase(const_iterator first, const_iterator last);  
2     // Erase from this string a substring defined by the specified pair of  
3     // 'first' and 'last' iterators within this string. Return an iterator  
4     // providing modifiable access to the character at the 'last' position  
5     // prior to erasing or 'end()' if no such character exists.
```

## WHAT THE FUNCTION RETURNS (CONT'D.)

- Unnecessary if the return type is void (obviously)
- Sometimes combined with "what the function does"

```
1 bool string_view::starts_with(string_view subview) const noexcept;  
2     // Return 'true' if this view starts with the specified 'subview', and  
3     // 'false' otherwise.
```

## ESSENTIAL BEHAVIOR

- Collateral effects and other consequences essential to the functionality

```
1 iterator erase(const_iterator first, const_iterator last);
2     // Erase from this string a substring defined by the specified pair of
3     // 'first' and 'last' iterators within this string. Return an iterator
4     // providing modifiable access to the character at the 'last' position
5     // prior to erasing. If no such character exists, return 'end()'.
6     // This method invalidates existing iterators pointing to 'first' or a
7     // subsequent position.
```

- Could include
  - Thread safety
  - Complexity guarantees
  - Iterator stability guarantees
  - Etc., etc., etc....

## PRECONDITIONS

- The behavior is undefined *unless* ...
  - Except when it would lead to a double-negative
- The *unless* form to use the same expression in the documentation and a contract-checking statement

```
1 iterator erase(const_iterator first, const_iterator last);
2     // Erase from this string a substring defined by the specified pair of
3     // 'first' and 'last' iterators within this string. Return an iterator
4     // providing modifiable access to the character at the 'last' position
5     // prior to erasing. If no such character exists, return 'end()'.
6     // This method invalidates existing iterators pointing to 'first' or a
7     // subsequent position. The behavior is undefined unless 'first' and
8     // 'last' are both within the range '[cbegin() .. cend()]' and 'first
9     // <= last'.
```

## BENEFITS OF *UNLESS*

- The *unless* form to use the same expression in the documentation and a contract-checking statement

```
1 void sqrt(double value)
2     // [...] The behavior is undefined unless '0 <= value'.
3 {
4     assert(0 <= value);
5     // [...]
6 }
```

## MORE BENEFITS OF *UNLESS*

- Multiple preconditions are additive
- Multiple assertions
- No need to apply DeMorgan's Laws

```
1 void Square::setDimensions(double width, double height)
2     // [...] The behavior is undefined unless
3     // '0 <= width' and '0 <= height'. [...]
4 {
5     assert(0 <= width);
6     assert(0 <= height);
7     // [...] Implementation
8 }
```

## PRECONDITION ORDER

- It is important to have a consistent order of preconditions
  - `void Object::func(T a, U b, V c);`
  - `P1()` - preconditions on global or object state
  - `P2(a)`, `P3(b)`, `P4(c)` - preconditions on individual arguments, in order
  - `P5(a, b)`, `P6(a, c)`, `P7(b, c)`, `P8(a, b, c)` - preconditions on relationships between arguments, in order

## ADDITIONAL NOTES

- Supplementary information potentially useful to the caller
- Aspects of behavior derivable from the contract, but not necessarily obvious

```
1 CHAR_TYPE *basic_string::data() noexcept;  
2     // [...] Note that any call to the string destructor or any of its  
3     // manipulators invalidates the returned pointer.
```

```
1 basic_string::operator basic_string_view<CHAR_TYPE, CHAR_TRAITS>() const noexcept;  
2     // [...] Note that this conversion operator can be invoked implicitly  
3     // (e.g., during argument passing).
```



## WHY WE CHOSE THIS STYLE?

- Conciseness
- Rigid structure allows spotting issues
- Human-oriented, not tool-oriented
- But there is a wide variety of documentation styles and frameworks
  - Doxygen
  - QDoc
  - ...
- Let's compare!

## EXAGGERATED EXAMPLE

```
1 /** * \brief Adds two numbers.
2  *
3  * This function takes two numbers, adds them, and then returns the result.
4  *
5  * \param x The first number to add.
6  * \param y The second number to add.
7  * \return The sum of the two numbers.
8  */
9 int add(int x, int y);
```

```
1 int add(int x, int y);
2     // Return the sum of the specified 'x' and 'y'. The behavior is
3     // undefined unless 'x + y' fits in the range '[INT_MIN, INT_MAX]'.
```

## REAL EXAMPLE

- JUCE framework
  - Excellently engineered
  - Excellently documented
  - 5k stars and 1.4k forks
- Goal of the examples that follow
  - Not to throw shade on JUCE in any way!
  - To provide a fair comparison
    - I randomly chose one component (`ZipFile`) with large doc blocks

## REAL EXAMPLE (DOXYGEN)

```
1  /** Adds a stream to the list of items which will be added to the archive.
2
3      @param streamToRead this stream isn't read immediately - a pointer to the stream is
4                          stored, then used later when the writeToStream() method is called,
5                          and deleted by the Builder object when no longer needed, so be
6                          very careful about its lifetime and the lifetime of any objects on
7                          which it depends!
8                          This must not be null.
9
10     @param compressionLevel    can be between 0 (no compression) and 9 (max compression)
11     @param storedPathName      the partial pathname that will be stored for this file
12     @param fileModificationTime the timestamp that will be stored as the last
13                                 modification time of this entry
14 */
15 void addEntry (InputStream* streamToRead,
16               int          compressionLevel,
17               const String& storedPathName,
18               Time         fileModificationTime);
```

## REAL EXAMPLE (BDE)

```
1 void addEntry (InputStream* stream,
2               int          compressionLevel,
3               const String& partialPathName,
4               Time         fileModificationTime);
5 // Add the specified 'stream' to the list of items that will be added to
6 // the archive, compressed at the specified 'compressionLevel', and stored
7 // with the specified 'partialPathName' and 'fileModificationTime'. The
8 // behavior is undefined unless '0 <= compressionLevel <= 9' or if 'stream'
9 // is null. Note that the 'stream' isn't read immediately - a pointer to
10 // the stream is stored, then used later when the writeToStream() method is
11 // called, and deleted by the Builder object when no longer needed, so be
12 // very careful about its lifetime and the lifetime of any objects on which
13 // it depends! Also note that '0' indicates no compression and '9'
14 // indicates maximum compression.
```

## **SUMMARY: CONTRACT DOCUMENTATION**

- Document your contracts
- Specific style is not important, but...
  - Choose a style!
    - Adjust it based on your requirements and stylistic preferences
  - Make sure your style includes all aspects of contracts
- Be consistent!

`assert` **IS NOT ENOUGH**

Contract checking at scale

## assert IS NOT ENOUGH

- Behavior on failure is not configurable
- All checks are treated the same irrespective of their complexity
- Adding checks to old code or modifying checks is difficult
- Testing contract checks is difficult



## PROBLEM 1: assert BEHAVIOR IS NOT CONFIGURABLE

- C assert outputs implementation-specific diagnostic to `stderr` and calls `std::abort`
- Application owner might want different behavior
  - Log to a different destination with a different format
  - Throw an exception?
  - Spin, waiting to attach a debugger?
  - Etc.
- We can install `SIGABRT` signal handler, but
  - Very limited in what it can do
  - Can't discriminate between assert violation and other sources of abort
  - No information about the violation

## SOLUTION 1: CONFIGURABLE VIOLATION HANDLER

```
using ViolationHandler = void (*)(const ContractViolation&);  
ViolationHandler setViolationHandler(ViolationHandler handler);
```

```
class ContractViolation {  
    // ...  
public:  
    std::source_location location() const;  
    std::string_view comment() const;  
};
```

## OFF-THE-SHELF HANDLERS

- `failByAbort` (default) – log a helpful message and unconditionally abort (similar behavior to `C assert`)
- `failBySleep` – log a helpful message and spin in an infinite loop (to wait for attaching a debugger)
- `failByThrow` – throw an `AssertTestException` (used for negative testing)
  - We'll talk about this shortly

## CONTINUING VIOLATION HANDLERS?

- Should we allow the handler to return normally?
  - In many application contexts, it's ill advised
  - The program is broken
  - We would likely go into Language UB territory
- But we need continuation in some circumstances
  - We'll discuss it a bit later
- For now, let's prevent continuation

## CCS\_ASSERT

```
1  #ifdef CCS_ASSERT_IS_ACTIVE
2  #define CCS_ASSERT(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X}); \
6              std::abort(); \
7          } \
8      } while (false)
9  #else
10 #define CCS_ASSERT(X) (void)sizeof((!(X))?true:false)
11 #endif
```

`CCS_ASSERT_IS_ACTIVE` is defined for checked builds, but not for unchecked ones.

## CHECKS SHOULD NOT AFFECT CONTROL FLOW (PART II)

```
1  std::pair<std::string_view, std::string_view> decomposeValue(  
2      std::string_view value)  
3      // Decompose the specified floating point 'value' into integral and  
4      // fractional parts. Throw 'std::domain_error' unless 'value' conforms  
5      // to '^[0-9]+(\.[0-9]+)?$'.  
6  {  
7      ccs::ContractViolationHandlerGuard g([](auto&&...) {  
8          throw std::domain_error("Bad input");  
9      });  
10  
11     CCS_ASSERT(!value.empty());  
12  
13     std::string_view integral =  
14         value.substr(0, value.find_first_not_of("0123456789"));  
15  
16     CCS_ASSERT(!integral.empty());  
17  
18     value.remove_prefix(integral.size());  
19  
20     std::string_view fractional = value.empty() ? "" : [&] {  
21         CCS_ASSERT(value[0] == '.');  
22         value.remove_prefix(1);  
23         CCS_ASSERT(!value.empty());  
24         CCS_ASSERT(value.find_first_not_of("0123456789") == value.npos);  
25         return value;  
26     }();  
27
```

```
28     return std::pair{integral, fractional};  
29 }
```

In an unchecked build, all checks will be disabled!

**VERY BAD IDEA!**

## ADDITIONAL UTILITIES

- `CCS_ASSERT_INVOKE_HANDLER` – invoke currently installed violation handler and abort (in all build modes)
- Sometimes useful to avoid computing the predicate multiple times

```
1  enum E : int { ONE, TWO };
2  void f(E e)
3  {
4      CCS_ASSERT(e == ONE || e == TWO);
5
6      switch (e) {
7          case ONE: { /* ... */ } break;
8          case TWO: { /* ... */ } break;
9      }
10 }
```



## ADDITIONAL UTILITIES

- `CCS_ASSERT_INVOKE_HANDLER` – invoke currently installed violation handler and abort (in all build modes)
- Sometimes useful to avoid computing the predicate multiple times

```
1 enum E : int { ONE, TWO };
2 void f(E e)
3 {
4     switch (e) {
5         case ONE: { /* ... */ } break;
6         case TWO: { /* ... */ } break;
7         default: { CCS_ASSERT_INVOKE_HANDLER("Bad enumeration value"); }
8     }
9 }
```

## PROBLEM 2: SOME CHECKS ARE TOO EXPENSIVE

- Application owners need to balance performance vs. amount of checking
- Library developers need a convenient way of allowing that control
- Should each check carry a number expressing the amount of checking work relative to the useful work of the function?

```
double sqrt(double value) { CCS_ASSERT(0 <= value, 3); /* ... */ }
```

```
int negate(int value) { CCS_ASSERT(value != INT_MIN, 100); /* ... */ }
```

```
lower_bound(R&& r, const T& value) { CCS_ASSERT(is_sorted(r), size(r) * 100); /* ... */ }
```

In practice, it's too cumbersome

## SOLUTION 2: TWO CLASSES OF CHECKS

- `CCS_ASSERT` (default) – checks that take less computation than the useful work the function does
- `CCS_ASSERT_AUDIT` – checks that take more than that
  
- `CCS_ASSERT` and `CCS_ASSERT_AUDIT` is a minimal useful set
- Enterprises might choose to implement more levels
  - Super-light checks that are almost always enabled
  - Super-heavy checks that break computational complexity

## PARTIAL CHECKING OF COMPLEX PRECONDITIONS

```
1  template <class ForwardIt, class T>
2  ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value)
3  {
4      ForwardIt it;
5      typename std::iterator_traits<ForwardIt>::difference_type count, step;
6      count = std::distance(first, last);
7
8      while (count > 0) {
9          it = first;
10         step = count / 2;
11         std::advance(it, step);
12
13         if (*it < value) {
14             first = ++it;
15             count -= step + 1;
16         }
17         else
18             count = step;
19     }
20
21     return first;
22 }
```

# PARTIAL CHECKING OF COMPLEX PRECONDITIONS

```
1  template <class ForwardIt, class T>
2  ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value)
3  {
4      ForwardIt it;
5      typename std::iterator_traits<ForwardIt>::difference_type count, step;
6      count = std::distance(first, last);
7
8      CCS_ASSERT_AUDIT(std::is_sorted(first, last));
9
10     while (count > 0) {
11         it = first;
12         step = count / 2;
13         std::advance(it, step);
14
15         if (*it < value) {
16             first = ++it;
17             count -= step + 1;
18         }
19         else
20             count = step;
21     }
22
23     return first;
24 }
```

# PARTIAL CHECKING OF COMPLEX PRECONDITIONS

```
1  template <class ForwardIt, class T>
2  ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value)
3  {
4      ForwardIt it;
5      typename std::iterator_traits<ForwardIt>::difference_type count, step;
6      count = std::distance(first, last);
7
8      CCS_ASSERT_AUDIT(std::is_sorted(first, last));
9
10     while (count > 0) {
11         it = first;
12         step = count / 2;
13         CCS_ASSERT(!(*std::next(it, step) < *it));
14         std::advance(it, step);
15
16         if (*it < value) {
17             first = ++it;
18             count -= step + 1;
19         }
20         else
21             count = step;
22     }
23
24     return first;
25 }
```



## PROBLEM 3: DIFFICULT TO ADD/MODIFY CHECKS

- Adding checks to an existing, functional system is not easy
  - Contract violations might be present, but the system works
  - Adding contract checks might lead to unnecessary outages
    - "Benign" contract violations (they are still defects!)
    - Contract check itself might be incorrect
  - Example: `std::optional::operator*()`
    - Behavior is undefined unless `has_value()`
    - Existing code dereferenced null optionals of fundamental types
- Similarly, changing the level of an existing check might lead to the same consequences

## SOLUTION 3: A DIFFERENT KIND OF ASSERT

- Do not abort execution after violation handler has finished
  - Even if it returned normally

```
1  #ifdef CCS_REVIEW_IS_ACTIVE
2  #define CCS_REVIEW(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X}); \
6              /* std::abort(); */ \
7          } \
8      } while (false)
9  #else
10 #define CCS_REVIEW(X) (void)sizeof(!(X)?true:false)
11 #endif
```

## HANDLING A REVIEW FAILURE

- Even a single incorrect use can result in `CCS_REVIEW` flooding the log
  - Might even be worse than abort, hogging resources while doing almost no useful work
- To mitigate that, the violation handler must be able to differentiate between `CCS_ASSERT` and `CCS_REVIEW`
  - For example, to use exponential backoff
- Associate a *semantic* with each contract check

## CONTRACT SEMANTICS

```
1 enum class semantic {
2     ignore, // The contract check is ignored
3     enforce, // If the check fails, abort if the handler returns
4     observe, // If the check fails, continue if the handler returns
5 };
```

## CONTRACT SEMANTICS (CONT'D.)

```
1 class ContractViolation {
2     // ...
3     public:
4         std::source_location location() const;
5         std::string_view comment() const;
6     };
```

## CONTRACT SEMANTICS (CONT'D.)

```
1 class ContractViolation {
2     // ...
3     public:
4         std::source_location location() const;
5         std::string_view comment() const;
6         semantic semantic() const;
7 };
```

## CONTRACT SEMANTICS (CONT'D.)

```
1  #ifdef CCS_ASSERT_IS_ACTIVE
2  #define CCS_ASSERT(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X}); \
6              std::abort(); \
7          } \
8      } while (false)
9  #else
10 #define CCS_ASSERT(X) (void)sizeof((!(X))?true:false)
11 #endif
```

## CONTRACT SEMANTICS (CONT'D.)

```
1  #ifdef CCS_ASSERT_IS_ACTIVE
2  #define CCS_ASSERT(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X, semantic::enforce}); \
6              std::abort(); \
7          } \
8      } while (false)
9  #else
10 #define CCS_ASSERT(X) (void)sizeof((!(X))?true:false)
11 #endif
```



## CONTRACT SEMANTICS (CONT'D.)

```
1  #ifdef CCS_REVIEW_IS_ACTIVE
2  #define CCS_REVIEW(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X}); \
6          } \
7      } while (false)
8  #else
9  #define CCS_REVIEW(X) (void)sizeof((!(X))?true:false)
10 #endif
```

## CONTRACT SEMANTICS (CONT'D.)

```
1  #ifdef CCS_REVIEW_IS_ACTIVE
2  #define CCS_REVIEW(X) \
3      do { \
4          if (!(X)) { \
5              invokeHandler({std::source_location::current(), #X, semantic::observe}); \
6          } \
7      } while (false)
8  #else
9  #define CCS_REVIEW(X) (void)sizeof((!(X))?true:false)
10 #endif
```

```

1 void exponentialBackoffHandler(const ContractViolation& violation)
2 {
3     switch (violation.semantic()) {
4         case ccs::semantic::enforce: {
5             std::cerr << "Contract violation enforced at "
6                 << violation.location().file_name() << ':'
7                 << violation.location().line() << " '" << violation.comment()
8                 << "'\n";
9         } break;
10        case ccs::semantic::observe: {
11            unsigned int violationCount = [&]() {
12                static std::map<std::pair<std::string, int>, unsigned int> counts;
13                static std::mutex countsMutex;
14
15                std::scoped_lock guard(countsMutex);
16                return ++counts[std::pair(violation.location().file_name(),
17                    violation.location().line())];
18            }();
19
20            if (std::has_single_bit(violationCount)) {
21                std::cerr << "Contract violation observed at "
22                    << violation.location().file_name() << ':'
23                    << violation.location().line() << " '"
24                    << violation.comment()
25                    << "' - Hit count: " << violationCount << '\n';
26            }
27        } break;
28    }

```



## CCS\_REVIEW LIFE CYCLE

- *New check in existing code or level change for existing check*
  - Add CCS\_REVIEW/CCS\_REVIEW\_AUDIT
  - Observe and fix any violations for a sufficient amount of time (e.g., 3 months)
  - Once no more violations are detected, convert to a CCS\_ASSERT/CCS\_ASSERT\_AUDIT
- 
- Abort for review violations in the handler used in development/testing environment

## EXAMPLE: CHANGING ASSERTION LEVEL

```
1 TYPE& optional::operator*()
2     // [...] The behavior is undefined unless 'has_value()'. [...]
3 {
4     CCS_ASSERT_AUDIT(has_value());
5
6     // [...] Implementation
7 }
```

## STEP 1: ADD A CCS\_REVIEW

```
1 TYPE& optional::operator*()
2     // [...] The behavior is undefined unless 'has_value()'. [...]
3 {
4     CCS_ASSERT_AUDIT(has_value());
5     CCS_REVIEW(has_value());
6
7     // [...] Implementation
8 }
```

## STEP 2: WAIT AND FIX ISSUES

```
1 TYPE& optional::operator*()
2     // [...] The behavior is undefined unless 'has_value()'. [...]
3 {
4     CCS_ASSERT_AUDIT(has_value());
5     CCS_REVIEW(has_value());
6
7     // [...] Implementation
8 }
```



### STEP 3: REPLACE WITH `CCS_ASSERT`

```
1 TYPE& optional::operator*()
2     // [...] The behavior is undefined unless 'has_value()'. [...]
3 {
4     CCS_ASSERT(has_value());
5
6     // [...] Implementation
7 }
```

## EXAMPLE: RUNNING PRODUCTION BUILDS WITH MORE EXPENSIVE CHECKS

- Scenario
  - We've been running production builds without a problem with `CCS_ASSERT` enabled
  - We want a subset of servers to do more checking (e.g., enabling `CCS_ASSERT_AUDIT`)
- Solution
  - Treat `CCS_ASSERT_AUDIT` as `CCS_REVIEW_AUDIT`
    - Keep `CCS_ASSERT` as before
- Similar to adding new checks in existing software

## EXAMPLE: NARROWING A CONTRACT

```
1 ConcurrentCache::ConcurrentCache(size_t lowWatermark, size_t highWatermark)
2     // [...] If 'lowWatermark > highWatermark', use the 'lowWatermark'
3     // value for both bounds. [...]
4 : d_lowWatermark(std::min(lowWatermark, highWatermark))
5 , d_highWatermark(highWatermark)
6 {
7 }
```

## STEP 1: ADD A CCS\_REVIEW

```
1 ConcurrentCache::ConcurrentCache(size_t lowWatermark, size_t highWatermark)
2     // [...] If 'lowWatermark > highWatermark', use the 'lowWatermark'
3     // value for both bounds. [...]
4 : d_lowWatermark(std::min(lowWatermark, highWatermark))
5 , d_highWatermark(highWatermark)
6 {
7     CCS_REVIEW(lowWatermark <= highWatermark);
8 }
```

## STEP 2: WAIT AND FIX ISSUES

```
1 ConcurrentCache::ConcurrentCache(size_t lowWatermark, size_t highWatermark)
2     // [...] If 'lowWatermark > highWatermark', use the 'lowWatermark'
3     // value for both bounds. [...]
4 : d_lowWatermark(std::min(lowWatermark, highWatermark))
5 , d_highWatermark(highWatermark)
6 {
7     CCS_REVIEW(lowWatermark <= highWatermark);
8 }
```

## STEP 3: CHANGE CONTRACT AND BEHAVIOR

```
1 ConcurrentCache::ConcurrentCache(size_t lowWatermark, size_t highWatermark)
2     // [...] The behavior is undefined unless 'lowWatermark <= highWatermark'.
3     // [...]
4 : d_lowWatermark(lowWatermark)
5 , d_highWatermark(highWatermark)
6 {
7     CCS_ASSERT(lowWatermark <= highWatermark);
8 }
```

## **PROBLEM 4: CONTRACT CHECKS ARE DIFFICULT TO TEST**

- Contract checks are code and ought to be unit tested
- Trigger a contract violation and verify that the check caught the issue (when enabled)
- How?
- Aborting handler would require death tests
  - Not supported in many testing frameworks
  - Significant performance penalty
  - Difficult to discriminate between assertion failure and other reason for a crash

## SOLUTION 4: AssertTest APPROACH

- Set up a throwing violation handler
  - Invoke the function under test out of contract
- Catch the exception
- Exception came from the wrong place?
  - Bug in a contract check
- Exception caught when there should have been none?
  - Bug in a contract check!
- No exception caught when there should have been one?
  - Believe it or not, bug in a contract check!



## EXAMPLE NEGATIVE TEST

```
1 void Socket::bind(const char* address,
2                 int      portNumber)
3 {
4     CCS_ASSERT(address);
5     CCS_ASSERT_AUDIT(isValidAddress(address));
6     CCS_ASSERT(0 <= portNumber && portNumber <= 65535);
7
8     // [...] Implementation
9 }
```

```
1 void doNegativeTest()
2 {
3     ccs::AssertTestHandlerGuard hG;
4
5     Socket socket;
6
7     CCS_ASSERTTEST_AUDIT_FAIL(socket.bind("256.0.0.1", 80));
8     CCS_ASSERTTEST_AUDIT_FAIL(socket.bind("127.0.0.1.", 80));
9     CCS_ASSERTTEST_AUDIT_FAIL(socket.bind("gibberish", 80));
10
11     CCS_ASSERTTEST_FAIL(socket.bind("127.0.0.1", -1));
12     CCS_ASSERTTEST_FAIL(socket.bind("127.0.0.1", 65536));
```

## CONTRACT CHECKS AND `noexcept`

- Wouldn't `noexcept` get in the way?
  - Yes, yes it would
- That's why we follow the Lakos Rule
  - Do ***not*** put `noexcept` on narrow-contract functions
  - See the original paper **N3279** and the upcoming **P2837** (May 15th)
- Alternatives are suboptimal
  - Death tests are not available in many testing frameworks
  - `setjmp/longjmp` – UB if jumping over anything non-trivial
    - Whether stack unwinding happens depends on the platform
  - `noexcept` active only in unchecked builds – not testing the actual production code

*Having thought more and having grown wiser, `_NOEXCEPT_DEBUG` was a horrible decision. It was viral, it didn't cover all the cases it needed to, and it was observable to the user – at worst changing the behavior of their program.*

– Eric Fiselier on the libcpp **changeset** switching to death testing (Mar 8, 2019)

See the upcoming paper **"P2834: Semantic Stability Across Contract-Checking Build Modes"** (May 15th)

## SUMMARY: CONTRACT CHECKING AT SCALE

- Provide means to set custom violation handler
  - Owner of main can choose the violation handling strategy
  - Be careful with allowing potentially continuing handlers
- Differentiate assertions by relative complexity
  - `CCS_ASSERT`, `CCS_ASSERT_AUDIT`
  - Easy categorization for the function author
  - Easy control of performance versus the amount of checking for the owner of `main`
- Provide mechanism to add new checks to old code
  - `CCS_REVIEW`
- Unit-test your contract checks!
  - Do **not** put `noexcept` on narrow-contract functions

## CONCLUSIONS

## CONCLUSIONS

- Carefully define the contracts for your functions
  - Minimal but complete
  - Leave behavior undefined for inputs that do not make semantic sense
- Defend against caller misuse using contract checks
- Choose a documentation style and follow it consistently
  - Make sure to document all important aspects of a function contract

## CONCLUSIONS

- Sufficiently flexible contract-checking system is crucial at scale
  - Configurable violation handler
  - Have a parallel facility for introducing new checks and similar tasks (CCS\_REVIEW)
  - A few assertion levels for ease of creation (by library authors) and control (by application owners)
  - Unit-test contract checks
    - Keep in mind not putting `noexcept` on functions having narrow contracts

