Avoid Misuse of Contracts!

Rostislav (Slava) Khlebnikov
Bloomberg LP

CppCon 2019

Outline

- Quick review
  - Terminology
  - Design by Contract (DbC)
  - Defensive Programming (DP)
- Misuse scenarios
  - Control flow
  - Side effects
  - Input validation
  - Additional thoughts
- Conclusion

Terminology

Terminology

```
template <class InputIt, class Distance>
void advance(InputIt& iterator, Distance n);
    // Increment the specified iterator by n elements.  The behavior is
    // undefined if '0 > n' unless 'InputIt' meets the requirements of
    // 'LegacyBidirectionalIterator', and if the specified sequence of
    // increments or decrements would require that a non-incrementable iterator
    // (such as the past-the-end iterator) is incremented, or that a
    // non-decrementable iterator (such as the front iterator or the singular
    // iterator) is decremented.  Note that this function has constant
    // complexity if 'InputIt' meets the requirements of
    // 'LegacyRandomAccessIterator', and linear complexity with respect to 'n'
    // otherwise.
```

Terminology: Precondition

```
template <class InputIt, class Distance>
void advance(InputIt& iterator, Distance n);
    // Increment the specified iterator by n elements.  The behavior is
    // undefined if '0 > n' unless 'InputIt' meets the requirements of
    // 'LegacyBidirectionalIterator', and if the specified sequence of
    // increments or decrements would require that a non-incrementable iterator
    // (such as the past-the-end iterator) is incremented, or that a
    // non-decrementable iterator (such as the front iterator or the singular
    // iterator) is decremented.  Note that this function has constant
    // complexity if 'InputIt' meets the requirements of
    // 'LegacyRandomAccessIterator', and linear complexity with respect to 'n'
    // otherwise.
```

Terminology: Postcondition

```
template <class InputIt, class Distance>
void advance(InputIt& iterator, Distance n);
    // Increment the specified iterator by n elements.  The behavior is
    // undefined if '0 > n' unless 'InputIt' meets the requirements of
    // 'LegacyBidirectionalIterator', and if the specified sequence of
    // increments or decrements would require that a non-incrementable iterator
    // (such as the past-the-end iterator) is incremented, or that a
    // non-decrementable iterator (such as the front iterator or the singular
    // iterator) is decremented.  Note that this function has constant
    // complexity if 'InputIt' meets the requirements of
    // 'LegacyRandomAccessIterator', and linear complexity with respect to 'n'
    // otherwise.
```

Terminology: Essential Behavior

```
template <class InputIt, class Distance>
void advance(InputIt& iterator, Distance n);
    // Increment the specified iterator by n elements.  The behavior is
    // undefined if '0 > n' unless 'InputIt' meets the requirements of
    // 'LegacyBidirectionalIterator', and if the specified sequence of
    // increments or decrements would require that a non-incrementable iterator
    // (such as the past-the-end iterator) is incremented, or that a
    // non-decrementable iterator (such as the front iterator or the singular
    // iterator) is decremented.  Note that this function has constant
    // complexity if 'InputIt' meets the requirements of
    // 'LegacyRandomAccessIterator', and linear complexity with respect to 'n'
    // otherwise.
```

Terminology: Narrow and Wide Contracts

Terminology: Narrow and Wide Contracts

- If a function has preconditions, it has a ***narrow contract***
  - `std::vector::operator[]()`
  - `std::vector::front()`

Terminology: Narrow and Wide Contracts

- If a function has preconditions, it has a ***narrow contract***
  - `std::vector::operator[]()`
  - `std::vector::front()`

- If a function has ***no*** preconditions, it has a ***wide contract***
  - `std::vector::push_back()`
  - `std::vector::size()`

Design by Contract

Design by Contract

```
double sqrt(double value);
```

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?
c. Return NaN?

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?
c. Return NaN?
d. Change the return type?

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?
c. Return NaN?
d. Change the return type?
   - optional<double>

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?
c. Return NaN?
d. Change the return type?
   - optional<double>
   - expected<double, SqrtError>

Design by Contract

```
double sqrt(double value);
```

- What to do if somebody passes us a negative value?

a. Throw an exception?
b. Return 0?
c. Return NaN?
d. Change the return type?
   - optional<double>
   - expected<double, SqrtError>
e. Something else?

Drawbacks of Artificially Wide Contracts (1/2)

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`
- Efficiency

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`
- Efficiency
  - Some preconditions break our algorithmic complexity guarantees
    - E.g., the range is sorted for `std::lower_bound`

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`
- Efficiency
  - Some preconditions break our algorithmic complexity guarantees
    - E.g., the range is sorted for `std::lower_bound`
  - Even if they don't it might be wasted effort

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`
- Efficiency
  - Some preconditions break our algorithmic complexity guarantees
    - E.g., the range is sorted for `std::lower_bound`
  - Even if they don't it might be wasted effort
  - Checks cannot be elided ***ever***

Drawbacks of Artificially Wide Contracts (1/2)

- Feasibility
  - Some preconditions very hard or impossible to check
    - E.g., that comparator defines a strict weak ordering for `std::sort`
- Efficiency
  - Some preconditions break our algorithmic complexity guarantees
    - E.g., the range is sorted for `std::lower_bound`
  - Even if they don't it might be wasted effort
  - Checks cannot be elided ***ever***
  - We pay the cost of checking upon ***every*** call

Drawbacks of Artificially Wide Contracts (2/2)

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects
- Maintainability

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects
- Maintainability
  - Both our code and our caller's code is more complex

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects
- Maintainability
  - Both our code and our caller's code is more complex
  - Additional testing is also required!

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects
- Maintainability
  - Both our code and our caller's code is more complex
  - Additional testing is also required!
- Extensibility

Drawbacks of Artificially Wide Contracts (2/2)

- Reliability
  - Artificially wide contracts tend to ***mask*** defects
- Maintainability
  - Both our code and our caller's code is more complex
  - Additional testing is also required!
- Extensibility
  - Once the decision was made, we can't turn back (easily)

Design by Contract

- Leave the behavior undefined!

Design by Contract

- Leave the behavior undefined!

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

Design by Contract

- Leave the behavior undefined!

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

- Design by contract: “If you give me valid input, I will behave as advertised; otherwise all bets are off!”
  - *John Lakos “Defensive Programming Done Right”, CppCon 2014*

Defensive Programming

Library vs Language Undefined Behavior

- What happens ***when*** a function is called out-of-contract?

Library vs Language Undefined Behavior

- What happens ***when*** a function is called out-of-contract?

```
size_t strlen(const char *str)
{
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Library vs Language Undefined Behavior

- What happens ***when*** a function is called out-of-contract?

```
size_t strlen(const char *str)
{
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Language UB might be here

Library vs Language Undefined Behavior

- What happens ***when*** a function is called out-of-contract?

```
size_t strlen(const char *str)
{
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Library (but not language) UB might have occurred here

Language UB might be here

Defensive checks

Defensive checks

- Undefined behavior is **undefined**!
  - We can do whatever we want if preconditions are violated!

Defensive checks

- Undefined behavior is **undefined**!
  - We can do whatever we want if preconditions are violated!

- We choose to help by checking preconditions

Defensive checks

- Undefined behavior is **undefined**!
  - We can do whatever we want if preconditions are violated!

- We choose to help by checking preconditions

- Do we have to ***always*** check?

Defensive checks

- Undefined behavior is **undefined**!
  - We can do whatever we want if preconditions are violated!

- We choose to help by checking preconditions

- Do we have to ***always*** check?
- Do we have to check ***everything***?

Defensive checks

- Undefined behavior is **undefined**!
  - We can do whatever we want if preconditions are violated!

- We choose to help by checking preconditions

- Do we have to ***always*** check?
- Do we have to check ***everything***?
- **NO!**

Defensive Checks

```
size_t strlen(const char *str)

{



    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Defensive Checks

```
size_t strlen(const char *str)

{
    assert(str);


    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Defensive Checks

```
size_t strlen(const char *str)

{
    assert(str);
    BSLS_ASSERT(str);

    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Defensive Checks

```
size_t strlen(const char *str)

{
    assert(str);
    BSLS_ASSERT(str);
    Expects(str);
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Defensive Checks

```
size_t strlen(const char *str)
    [[pre: str]]
{
    assert(str);
    BSLS_ASSERT(str);
    Expects(str);
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

Defensive Checks

```
size_t strlen(const char *str)
    [[pre: str]]
{
    assert(str);
    BSLS_ASSERT(str);
    Expects(str);
    size_t count = 0;
    while (str[count])
        ++count;
    return count;
}
```

- Can be turned off
  - Without changing program behavior

The Principles

The Principles

- In a defect-free program, no contracts should be violated
  - A violated contract is a bug!

The Principles

- In a defect-free program, no contracts should be violated
  - A violated contract is a bug!
- Removing contract checks should not affect program's *essential* behavior
  - Except, perhaps, its runtime performance

(Mis)using Contract Checks for Control Flow

Meaning of a Contract Check

```
double sqrt(double value)
    [[pre: 0 <= value]]
{
    // ...
}
```

Meaning of a Contract Check

```
double sqrt(double value)
    [[pre: 0 <= value]]
{
    // ...
}
```

What does this express?

Meaning of a Contract Check

```
double sqrt(double value)
    [[pre: 0 <= value]]
{
    // ...
}
```

What does this express?

- A hope that value is non-negative? (But we can recover?)

Meaning of a Contract Check

```
double sqrt(double value)
    [[pre: 0 <= value]]
{
    // ...
}
```

What does this express?

- A hope that value is non-negative? (But we can recover?)
- A statement of absolute truth – value will NEVER be negative?

Meaning of a Contract Check

```
double sqrt(double value)
    [[pre: 0 <= value]]
{
    // ...
}
```

What does this express?

- A hope that value is non-negative? (But we can recover?)
- A statement of absolute truth – value will NEVER be negative?
- **A statement of what *should* be true in a *correct* program!**

(Mis)using Contract Checks for Control Flow

```
class MyIntVector {
    int* d_data;
    // . . .
public:
    // . . .

    int operator[](int index) const
    {




        return d_data[index];
    }
};
```

(Mis)using Contract Checks for Control Flow

```
class MyIntVector {
    int* d_data;
    // . . .
public:
    // . . .

    int operator[](int index) const
    {
        [[assert: 0 <= index]];
        [[assert: index < size()]];

        return d_data[index];
    }
};
```

(Mis)using Contract Checks for Control Flow

```
class MyIntVector {
    int* d_data;
    // . . .
public:
    // . . .

    int operator[](int index) const
    {
        [[assert: 0 <= index]];
        [[assert: index < size()]];

        return d_data[index];
    }
};
```

```
class Data {
    MyIntVector d_data;
  public:
    std::optional<int> getValue(int index) const
    {

            return d_data[index];



    }
};
```

(Mis)using Contract Checks for Control Flow

```
class MyIntVector {
    int* d_data;
    // . . .
public:
    // . . .

    int operator[](int index) const
    {
        [[assert: 0 <= index]];
        [[assert: index < size()]];

        return d_data[index];
    }
};
```

```
class Data {
    MyIntVector d_data;
  public:
    std::optional<int> getValue(int index) const
    {
        try {
            return d_data[index];
        } catch (const ContractViolationException& ex) {
            return std::nullopt;
        }
    }
};
```

(Mis)using Contract Checks for Control Flow

```
class MyIntVector {
    int* d_data;
    // . . .
public:
    // . . .

    int operator[](int index) const
    {
        [[assert: 0 <= index]];
        [[assert: index < size()]];

        return d_data[index];
    }
};
```

```
class Data {
    MyIntVector d_data;
  public:
    std::optional<int> getValue(int index) const
    {
        try {
            return d_data[index];
        } catch (const ContractViolationException& ex) {
            return std::nullopt;
        }
    }
};

int main() {
    set_violation_handler(
        [](auto&&) { throw ContractViolationException{}; }
    );

    // Use 'Data'...
}
```

Misuse Scenarios: Side Effects

Side Effects: In Predicates

- It’s a bug if inserting `value` into `setOfIntegers` fails

Side Effects: In Predicates

- It's a bug if inserting `value` into `setOfIntegers` fails
- Is this a good way to check it?

```
[[assert: setOfIntegers.insert(value).second]];
```

Side Effects: In Predicates

- It's a bug if inserting `value` into `setOfIntegers` fails
- Is this a good way to check it?

```
[[assert: setOfIntegers.insert(value).second]];

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

Side Effects: In Predicates

```
class EncryptedStore {
    std::map<int, int> d_map;

  public:

    void corruptValueAt(int index)
        [[pre: !isCorrupted(d_map[index])]];
};
```

Is this a good check?

Side Effects: Not Affecting Essential Behavior

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {



        return d_fields.find(std::string(name)) != d_fields.end();
    }





};
```

Side Effects: Not Affecting Essential Behavior

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {



        return d_fields.find(std::string(name)) != d_fields.end();
    }

    void addField(std::string_view name, std::string_view value)
        [[pre: !contains(name)]]
    {
        // . . .
    }
};
```

Side Effects: Not Affecting Essential Behavior

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {



        return d_fields.find(std::string(name)) != d_fields.end();
    }

    void addField(std::string_view name, std::string_view value)
        [[pre: !contains(name)]]
    {
        // . . .
    }
};
```

Which side effects happen in the predicate? Is that OK?

Side Effects: Not Affecting Essential Behavior

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {



        return d_fields.find(std::string(name)) != d_fields.end();
    }

    void addField(std::string_view name, std::string_view value)
        [[pre: !contains(name)]]
    {
        // . . .
    }
};
```

Potentially allocates

Which side effects happen in the predicate? Is that OK?

Side Effects: Not Affecting Essential Behavior

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {
        LOG_TRACE << "Checking whether '" << name << "' is present among "
                  << d_fields.size() << " fields.";

        return d_fields.find(std::string(name)) != d_fields.end();
    }

    void addField(std::string_view name, std::string_view value)
        [[pre: !contains(name)]]
    {
        // . . .
    }
};
```

Writes to log

Potentially allocates

Which side effects happen in the predicate? Is that OK?

Misuse Scenarios: Input Validation

Input Validation

Input Validation

Input Validation

- What is “input”?
  - Any data coming into the system from untrusted sources
    - Command line
    - File
    - Over-the-wire
    - …

Input Validation

- What is “input”?
  - Any data coming into the system from untrusted sources
    - Command line
    - File
    - Over-the-wire
    - …
- Generally, anything coming from outside the *application envelope*

Application Envelope

- How large is the application envelope?
  - Depends on the application
    - Compiled source code
    - Config files
    - Verified remote application
    - …

Application Envelope

- How large is the application envelope?
  - Depends on the application
    - Compiled source code
    - Config files
    - Verified remote application
    - …
- Rule of thumb:
  If something is ***NOT*** a part of your testing process, it is ***DEFINITELY NOT*** a part of the application envelope

Example

Example

- Proof-of-concept (throwaway) translation application
  - Only support translation of “Hello” and “Goodbye”

Example

- Proof-of-concept (throwaway) translation application
  - Only support translation of “Hello” and “Goodbye”

- Name of file containing the word to translate supplied on command line

Example

- Proof-of-concept (throwaway) translation application
  - Only support translation of “Hello” and “Goodbye”
- Name of file containing the word to translate supplied on command line
- Output the translation to standard output

Input Validation: PoC application

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);









}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);




    std::string greeting;
    in  >> greeting;




}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);




    std::string greeting;
    in  >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);



    std::string greeting;
    in  >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);




    std::string greeting;
    in  >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);



}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);



    std::string greeting;
    in  >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);



    std::string greeting;
    [[assert: in >> greeting]];


    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);



    std::string greeting;
    [[assert: in >> greeting]];


    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
```

BAD!

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

GOOD!

```
int main(int argc, const char* argv[])
{


    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre:  fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre:  fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{


    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

GOOD!

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

OK

OK

OK

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

OK

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: PoC application

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

GOOD!

OK

OK

OK

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

OK

GOOD!

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)

{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
```

GOOD!

```
[[pre: fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{


    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {


        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre:  fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{



    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre:  fileName]]
{
    std::ifstream in(fileName);



    std::string greeting;
    in >> greeting;

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {

        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{


    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

GOOD!

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

BAD!

BAD!

BAD!

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

BAD!

I. No contracts should be violated.
II. Removing contracts should not affect program's *essential* behavior.

Input Validation: Production Desktop App

```
enum class Greeting {
    HELLO,
    GOODBYE
};

Greeting loadGreeting(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);

    [[assert: in]];

    std::string greeting; in >> greeting;
    [[assert: in]];

    if (greeting == "Hello") {
        return Greeting::HELLO;
    } else {
        [[assert: greeting == "Goodbye"]];
        return Greeting::GOODBYE;
    }
}
```

GOOD!

BAD!

BAD!

BAD!

```
int main(int argc, const char* argv[])
{
    [[assert: argc == 2]];

    Greeting gt = loadGreeting(argv[1]);
    if (gt == Greeting::HELLO) {
        std::cout << "Bonjour";
    } else {
        [[assert: gt == Greeting::GOODBYE]];
        std::cout << "Au revoir";
    }
}
```

BAD!

GOOD!

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)

{
    std::ifstream in(fileName);

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

        result.emplace(from, to);

    }
}
```

```
int main(int argc, const char* argv)
{

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)

{
    std::ifstream in(fileName);

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

        [[assert: result.emplace(from, to).second]];
    }
}
```

```
int main(int argc, const char* argv)
{

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)

{
    std::ifstream in(fileName);

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

    [[assert: result.emplace(from, to).second]];
    }
}
```

BAD!

```
int main(int argc, const char* argv)
{

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);


    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

         result.emplace(from, to);


    }
}
```

```
int main(int argc, const char* argv)
{



    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

GOOD!

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);


    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

         result.emplace(from, to);


    }
}
```

```
int main(int argc, const char* argv)
{



    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

        result.emplace(from, to);

    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

        result.emplace(from, to);

    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;

        result.emplace(from, to);

    }
}
```

OK

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

OK

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                  -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                  -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
OK      [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
OK      [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());

        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                  -> HttpResponse {
        auto [it, found] = dict.find(req.data());
        [[assert : found]];
        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());
        [[assert : found]];
        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

BAD!

Input Validation: Containerized Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

GOOD!

OK

OK

OK

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());
        [[assert : found]];
        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

OK

BAD!

Input Validation: Isolated High Throughput Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                  -> HttpResponse {
        auto [it, found] = dict.find(req.data());
        [[assert : found]];
        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

Input Validation: Isolated High Throughput Service

```
auto loadDictionary(const char* fileName)
    [[pre: fileName]]
{
    std::ifstream in(fileName);
    [[assert: in]];

    std::unordered_map<std::string,
                       std::string> result;
    std::string from, to;
    while (true) {
        in >> from;
        if (!in) { return result; }
        in >> to;
        [[assert: in]];
        auto [_, ok] = result.emplace(from, to);
        [[assert: ok]];
    }
}
```

```
int main(int argc, const char* argv)
{
    [[assert: argc == 2]];

    Dictionary dict = loadDictionary(argv[1]);

    HttpServer server(80);
    server.listen([&](const HttpRequest& req)
                                    -> HttpResponse {
        auto [it, found] = dict.find(req.data());
        [[assert : found]];
        return makeResponse(HttpStatus::k_OK, it->second);
    });
}
```

OK

Additional Thoughts

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express

```
void mutex::unlock();
    // [...] The behavior is undefined unless
    // the calling thread currently owns the lock on this mutex.
```

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details

```
DataKey createKey() { return d_nextKey++; }

void Registry::setData(DataKey key)
    [[pre: key < d_nextKey]];
```

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details
- Sometimes it’s worth considering widening contract

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details
- Sometimes it's worth considering widening contract

```
AST build_AST(std::string_view sourceCode)
    [[pre: is_valid_cpp_code(sourceCode)]];
```

Additional Thoughts

- Contract checks do not replace thorough testing
  - They merely complement it
- Contract checks do not replace proper documentation
  - Some aspects of documentation are difficult to express
  - Checks often need to access (and expose) implementation details
- Sometimes it’s worth considering widening contract

```
AST build_AST(std::string_view sourceCode)
    [[pre: is_valid_cpp_code(sourceCode)]];


expected<AST, ParseError> build_AST(std::string_view sourceCode);
```

Conclusion

Conclusion

- No matter the *contract-checking facility*:

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior
  - Eliminate effect on essential behavior

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior
  - Eliminate effect on essential behavior
- Don't confuse input validation with contract checking!

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior
  - Eliminate effect on essential behavior
- Don't confuse input validation with contract checking!
- Use engineering sense

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior
  - Eliminate effect on essential behavior
- Don't confuse input validation with contract checking!
- Use engineering sense
  - Which side effects are acceptable?

Conclusion

- No matter the *contract-checking facility*:
  - No contract should be violated
  - Removing contracts should not affect program's essential behavior

- Beware of side effects in predicates
  - Minimize effect on non-essential behavior
  - Eliminate effect on essential behavior
- Don't confuse input validation with contract checking!
- Use engineering sense
  - Which side effects are acceptable?
  - What constitutes your *application envelope*?

Questions?