Contract Use: Past, Present, and Future

Joshua Berne - jberne4@bloomberg.net

2019-09-18

Copyright Notice

1 Introduction

Who am I?

- Software developer all century
- I have a purple house.

- First time presenting at CppCon
- First time presenting at a Conference

Who am I?

- Software developer all century
- I have a purple house.

- First time presenting at CppCon
- First time presenting at a Conference (Be gently please)

Who am I?

- Bloomberg LP since 2017
- Joined BDE team in 2018
- Contract checking and deployment with `BSLS_REVIEW`
- WG21 participation to make contracts better
- SG21 participation with same goal

English Contracts

Contracts are an agreement between two parties

English Contracts

Software contracts are an agreement between a library writer and client

English Contracts

Function contracts can be rendered in english

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

English Contracts

Describe what a function will do

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

English Contracts

Describe what behavior is not supported

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

Undefined Behavior

undefined behavior
behavior for which this document imposes no requirements

N4830 - Working Draft, Standard for Programming Language C++

Undefined Behavior

undefined behavior
behavior for which this document imposes no requirements

N4830 - Working Draft, Standard for Programming Language C++

library undefined behavior
behavior for which a library contract provides no guarantees

John Lakos - CppCon 2014

Undefined Behavior

language undefined behavior
behavior for which this document imposes no requirements

N4830 - Working Draft, Standard for Programming Language C++

library undefined behavior
behavior for which a library contract provides no guarantees

John Lakos - CppCon 2014

English Contracts

Describe what behavior is not supported

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

English Contracts

Preconditions

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

English Contracts

Postconditions

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

English Contracts

Essential Behavior

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

Violating a contract is a bug

Violating a contract is a bug

Bugs are contract violations

Violating a contract is a bug

Bugs are contract violations

Possibly a contract no one wrote down

What can be checked?

- Parts of the english contract might be checkable with standard C++ expressions.

What can be checked?

- Parts of the english contract might be checkable with standard C++ expressions.
- Parts might have readbale representations that cannot be implemented

What can be checked?

- Parts of the english contract might be checkable with standard C++ expressions.
- Parts might have readbale representations that cannot be implemented
- Parts might be statements beyond the scope of a single function execution

In code contracts

Some parts can be rendered with code

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

In code contracts

Simple boolean predicates

```
begin != nullptr
end != nullptr
begin <= end
```

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

In code contracts

Predicates about returned value

```
return_val >= begin
return_val <= end
return_val == end  *return_val >= val
```

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

In code contracts

Hard to check things

```
is_sorted(begin,end)
```

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

In code contracts

Uncheckable things?

```
is_reachable_from(begin,end)
```

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

In code contracts

Properties of repeated execution

```
??????????????????
```

```
T* binsearch(T*begin, T*end, const T& val);
  // Return a pointer to an element between the
  // specified 'begin' and 'end' that is greater
  // than or equal to the specified 'val', or 'end'
  // if no such value exists.  This function will
  // perform no more than log(distance(begin,end))
  // comparisons.  The  behavior is undefined
  // unless '[begin,end)' is a contiguous sorted
  // range.
```

Proven contracts

- Prove software correctness

Proven contracts

- Prove software correctness
- Encode contracts completely

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions
  - All essential behavior

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions
  - All essential behavior
- Statically prove everything

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions
  - All essential behavior
- Statically prove everything
  - For each function, prove postconditions and essential behavior

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions
  - All essential behavior
- Statically prove everything
  - For each function, prove postconditions and essential behavior
  - Use called functions contracst in proofs of larger functions

Proven contracts

- Prove software correctness
- Encode contracts completely
  - All preconditions
  - All postconditions
  - All essential behavior
- Statically prove everything
  - For each function, prove postconditions and essential behavior
  - Use called functions contracst in proofs of larger functions
- PROFIT

Dream Benefit #1 - Less Bugs

Dream Benefit #1 - Less Bugs

- Compiler identifies all violated contracts

Dream Benefit #1 - Less Bugs

- Compiler identifies all violated contracts
- Edge cases must be throught through

Dream Benefit #1 - Less Bugs

- Compiler identifies all violated contracts
- Edge cases must be throught through
- All assumptions are captured in compiled code

Dream Benefit #1 - Less Bugs

- Compiler identifies all violated contracts
- Edge cases must be throught through
- All assumptions are captured in compiled code
- Mostly, if it compiles, it doesn't have bugs (contract violations)

Dream Benefit #1 - Less Bugs

- Compiler identifies all violated contracts
- Edge cases must be throught through
- All assumptions are captured in compiled code
- Mostly, if it compiles, it doesn't have bugs (contract violations)
- If there is a bug, contracts just need to be elaborated

Dream Benefit #2 - Less Heat Generation

Dream Benefit #2 - Less Heat Generation

- No need for any checks

Dream Benefit #2 - Less Heat Generation

- No need for any checks
- More knowledge for the compiler

Dream Benefit #2 - Less Heat Generation

- No need for any checks
- More knowledge for the compiler
  - `__builtin_assume`

Dream Benefit #2 - Less Heat Generation

- No need for any checks
- More knowledge for the compiler
  - `__builtin_assume`
  - Removing excess branches

Dream Benefit #2 - Less Heat Generation

- No need for any checks
- More knowledge for the compiler
  - `__builtin_assume`
  - Removing excess branches
  - Vectorization/SIMD instructions

Dream Benefit #2 - Less Heat Generation

- No need for any checks
- More knowledge for the compiler
  - `__builtin_assume`
  - Removing excess branches
  - Vectorization/SIMD instructions
- Smaller code size

Realizing parts of the dream

- WARNING:

Realizing parts of the dream

- WARNING: MACROS INCOMING

Realizing parts of the dream

- WARNING: MACROS INCOMING
- How to leverage contracts without a language feature

Realizing parts of the dream

- WARNING: MACROS INCOMING
- How to leverage contracts without a language feature
- Bloomberg has been doing this for 15 years

Realizing parts of the dream

- WARNING: MACROS INCOMING
- How to leverage contracts without a language feature
- Bloomberg has been doing this for 15 years
- See the BDE open source repostory for the real implementation
  - `https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_assert.h`
  - `https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_review.h`

What do you do if you can't prove a contract is being followed?

What do you do if you can't prove a contract is being followed?

Experiment

Documenting expectations

- Initial benefit of contracts in code

Documenting expectations

- Initial benefit of contracts in code

```
#define ASSERT(X)
```

Documenting expectations

- Initial benefit of contracts in code
- Bloomberg specific naming

```
#define BSLS_ASSERT(X)
```

Documenting expectations

- Initial benefit of contracts in code
- Bloomberg specific naming
- Avoid code rot

```
#define BSLS_ASSERT(X) sizeof( (X)?true:false )
```

Documenting expectations

- Initial benefit of contracts in code
- Bloomberg specific naming
- Avoid code rot
- ... wish we had done that originally

```
#ifdef BSLS_ASSERT_VALIDATE_DISABLED_MACROS
#define BSLS_ASSERT(X) sizeof( (X)?true:false )
#else
#define BSLS_ASSERT(X)
#endif
```

Documenting expectations

- Initial benefit of contracts in code
- Bloomberg specific naming
- Avoid code rot
- ... wish we had done that originally
- ... or at least this to require a ;

```
#define BSLS_ASSERT(X) ((void)0)
```

Documenting expectations

- Initial benefit of contracts in code
- Bloomberg specific naming
- Avoid code rot
- ... wish we had done that originally
- ... or at least this to require a ;
- For simplicity

```
#define ASSERT(X)
```

Aborting on bugs

- Identifying violations would be nice

Aborting on bugs

- Identifying violations would be nice
- The safest thing to do is stop immediately

```
#define ASSERT(X) if (!(X)) { std::abort(); }
```

Aborting on bugs

- Identifying violations would be nice
- The safest thing to do is stop immediately
- Nice if `ASSERT(X)` needs a semicolon

```
#define ASSERT(X) do { if (!(X)) { std::abort(); } } while (false)
```

Aborting on bugs

- Identifying violations would be nice
- The safest thing to do is stop immediately
- Nice if `ASSERT(X)` needs a semicolon
- For simplicity

```
#define ASSERT(X) if (!(X)) { std::abort(); }
```

... only in some builds

- Checks of contracts are redundant if they're not broken

... only in some builds

- Checks of contracts are redundant if they're not broken
- NDEBUG might be a way to control enablement

```
#ifdef NDEBUG
#define ASSERT(X)
#else
#define ASSERT(X) if (!(X)) { std::abort(); }
#endif
```

... only in some builds

- Checks of contracts are redundant if they're not broken
- NDEBUG might be a way to control enablement
- This reminds me of something

```
#include <cassert>
#define ASSERT(X) assert(X)
```

... only in some builds

- Checks of contracts are redundant if they're not broken
- NDEBUG might be a way to control enablement
- This reminds me of something
- Separating out controls from behavior helps

```
#define ASSERT_IMP(X)          if (!(X)) { std::abort(); }
```

... only in some builds

- Checks of contracts are redundant if they're not broken
- NDEBUG might be a way to control enablement
- This reminds me of something
- Separating out controls from behavior helps

```
#define ASSERT_IMP(X)           if (!(X)) { std::abort(); }
#define ASSERT_DISABLED_IMP(X)
```

... only in some builds

- Checks of contracts are redundant if they're not broken
- NDEBUG might be a way to control enablement
- This reminds me of something
- Separating out controls from behavior helps

```
#define ASSERT_IMP(X)           if (!(X)) { std::abort(); }
#define ASSERT_DISABLED_IMP(X)

#ifdef ASSERT_LEVEL_ASSERT
#define ASSERT(X) ASSERT_IMP(X)
#else
#define ASSERT(X) ASSERT_DISABLED_IMP(X)
#endif
```

... but what happened?

- Aborting with no information sucks

```
#define ASSERT_IMP(X) if (!(X)) {                                    \
    /*POOF*/;                                                        \
    std::abort();                                                    \
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps

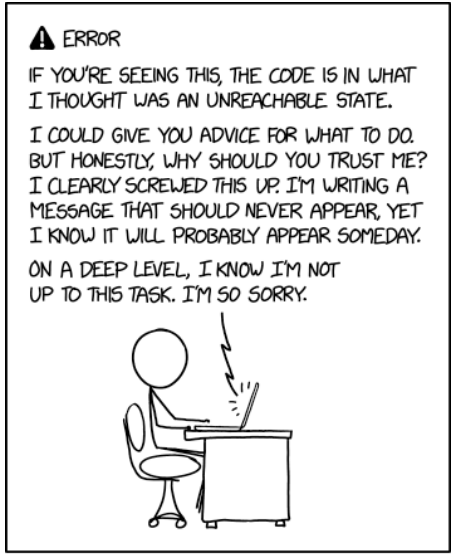
```
#define ASSERT_IMP(X) if (!(X)) {                                    \
    printf("ASSERTION FAILED!\n");                                   \
                                                                     \
    std::abort();                                                    \
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help

```
#define ASSERT_IMP(X) if (!(X)) {                                    \
    printf("ASSERTION FAILED (" __FILE__ ":%d): %s\n",               \
                                __LINE__, #X);                       \
    std::abort();                                                    \
}
```

What about this guy?



NEVER WRITE ERROR MESSAGES TIRED.

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that

```
#define ASSERT_IMP(X) if (!(X)) {                                          \
    bb::Assert::invoke_violation_handler(__FILE__, __LINE__, #X);          \
                                                                           \
    std::abort();                                                          \
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that

```
#define ASSERT_IMP(X) if (!(X)) {                                    \
    bb::assert_violation violation(__FILE__, __LINE__, #X); \
    bb:Assert::invoke_violation_handler(violation);                  \
    std::abort();                                                    \
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that
- Leave all behavior up to the violation handler

```
#define ASSERT_IMP(X) if (!(X)) {                                    \
    bb::assert_violation violation(__FILE__, __LINE__, #X);          \
    bb::Assert::invoke_violation_handler(violation);                 \
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that
- Leave all behavior up to the violation handler

```
void bb::Assert::invoke_violation_handler(
                                const bb::assert_violation &violation) {
    getViolationHandler()(violation);
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that
- Leave all behavior up to the violation handler

```
void xkcd::violationHandler(const bb::assert_violation &violation) {
  printf("Error\n");
  printf("If you're seeing this, the code is in what\n");
  printf("I thought was an unreachable state.");
  //...
}
```

... but what happened?

- Aborting with no information sucks
- Logging something helps
- The preprocessor can give us more help
- Delegating to a pluggable function helps that
- Leave all behavior up to the violation handler

```
int main() {
  bb::Assert::setViolationHandler(&xkcd::violationHandler);
  //..
}
```

... that doesn't work everywhere!

- The violation handler can notify in different ways

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things
  - `std::abort()`

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things
  - `std::abort()`
  - `while (true) {std::this_thread::sleep_for(std::chrono::years(1));}`

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things
  - `std::abort()`
  - `while (true) {std::this_thread::sleep_for(std::chrono::years(1));}`
  - `throw std:::exception("Oops?");`

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things
  - `std::abort()`
  - `while (true) {std::this_thread::sleep_for(std::chrono::years(1));}`
  - `throw std:::exception("Oops?");`
- ... or try to recover?

... that doesn't work everywhere!

- The violation handler can notify in different ways
  - Custom logging frameworks
  - GUI messages (abort, retry, fail?)
  - Hardware notifications
- ... do different things
  - `std::abort()`
  - `while (true) {std::this_thread::sleep_for(std::chrono::years(1));}`
  - `throw std:::exception("Oops?");`
- ... or try to recover?
- `main` gets to decide

Checking is slow!

- Checks use state already in cache, are often very fast

```
T* binsearch(T*begin, T*end, const T& val) {
  ASSERT(begin);
  ASSERT(end);
  ASSERT(begin < end)7
  //..
}
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that

```
T* binsearch(T*begin, T*end, const T& val) {
  ASSERT(is_sorted_range(begin,end));
  //..
}
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity

```
#define ASSERT_OPT(X) ...
#define ASSERT(X) ...
#define ASSERT_SAFE(X) ...
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient

```
[[ assert default : X ]];
[[ assert audit : X ]];
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
#if defined(ASSERT_LEVEL_NONE)    ? 1 : 0 \
  + defined(ASSERT_LEVEL_OPT)     ? 1 : 0 \
  + defined(ASSERT_LEVEL_ASSERT)  ? 1 : 0 \
  + defined(ASSERT_LEVEL_SAFE)    ? 1 : 0 \
  > 1
#error Multiple ASSERT_LEVEL macros defined
#endif
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
#if !defined(ASSERT_LEVEL_NONE)    \
 && !defined(ASSERT_LEVEL_OPT)     \
 && !defined(ASSERT_LEVEL_ASSERT)  \
 && !defined(ASSERT_LEVEL_SAFE)
#define ASSERT_LEVEL_ASSERT
#endif
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
#if defined(ASSERT_LEVEL_NONE)
#define ASSERT_OPT(X)  ASSERT_DISABLED_IMP(X)
#define ASSERT(X)      ASSERT_DISABLED_IMP(X)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
//..
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
//..
#elif defined(ASSERT_LEVEL_OPT)
#define ASSERT_OPT(X)  ASSERT_IMP(X)
#define ASSERT(X)      ASSERT_DISABLED_IMP(X)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
//..
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
//..
#elif defined(ASSERT_LEVEL_ASSERT)
#define ASSERT_OPT(X)  ASSERT_IMP(X)
#define ASSERT(X)      ASSERT_IMP(X)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
//..
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
//..
#elif defined(ASSERT_LEVEL_SAFE)
#define ASSERT_OPT(X)  ASSERT_IMP(X)
#define ASSERT(X)      ASSERT_IMP(X)
#define ASSERT_SAFE(X) ASSERT_IMP(X)
#endif
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
#if defined(ASSERT_LEVEL_OPT)     \
  defined(ASSERT_LEVEL_ASSERT) \
  defined(ASSERT_LEVEL_SAFE)
#define ASSERT_OPT(X) ASSERT_IMP(X)
#else
 // defined(ASSERT_LEVEL_NONE)
#define ASSERT_OPT(X) ASSERT_DISABLED_IMP(X)
#endif
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement

```
#if defined(ASSERT_LEVEL_ASSERT) \
  defined(ASSERT_LEVEL_SAFE)
#define ASSERT(X) ASSERT_IMP(X)
#else
 // defined(ASSERT_LEVEL_OPT)
 // defined(ASSERT_LEVEL_NONE)
#define ASSERT(X) ASSERT_DISABLED_IMP(X)
#endif
```

Checking is slow!

- Checks use state already in cache, are often very fast
- Algorithmic complexity can still ruin that
- 3 levels of complexity
- ... 2 levels probably sufficient
- Linear scale of enablement
- Bloomberg 2005-2018

```
#if defined(ASSERT_LEVEL_SAFE)
#define ASSERT_SAFE(X) ASSERT_IMP(X)
#else
 // defined(ASSERT_LEVEL_OPT)
 // defined(ASSERT_LEVEL_ASSERT)
 // defined(ASSERT_LEVEL_NONE)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
#endif
```

Choosing Levels in code

- Original Suggestion:

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower
- Current Suggestion:

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower
- Current Suggestion:
  - `OPT`: 0-0.5% absolutely critical and 0-impact

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower
- Current Suggestion:
  - `OPT`: 0-0.5% absolutely critical and 0-impact
  - `ASSERT`: 99% non $O(n)$-impacting

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower
- Current Suggestion:
  - `OPT`: 0-0.5% absolutely critical and 0-impact
  - `ASSERT`: 99% non $O(n)$-impacting
  - `SAFE`: 0.5-1% algorithmicly slow

Choosing Levels in code

- Original Suggestion:
  - `OPT`: 5% most critical tests
  - `ASSERT`: 90% tests <2x performance hit
  - `SAFE`: 5% anything slower
- Current Suggestion:
  - `OPT`: 0-0.5% absolutely critical and 0-impact
  - `ASSERT`: 99% non $O(n)$-impacting
  - `SAFE`: 0.5-1% algorithmicly slow
- Changing is hard

Choosing Levels in builds

- What we did

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted
  - Developement - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted
  - Developement - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Unit tests - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted
  - Developement - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Unit tests - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Beta testing - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted
  - Developement - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Unit tests - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Beta testing - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Production - `ASSERT_LEVEL_ASSERT`

Choosing Levels in builds

- What we did
  - Developement - `ASSERT_LEVEL_ASSERT`
  - Unit tests - `ASSERT_LEVEL_ASSERT` hopefully
  - Beta testing - `ASSERT_LEVEL_ASSERT_OPT`
  - Production - `ASSERT_LEVEL_ASSERT_OPT`
- What we wanted
  - Developement - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Unit tests - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Beta testing - `ASSERT_LEVEL_ASSERT` or `ASSERT_LEVEL_SAFE`
  - Production - `ASSERT_LEVEL_ASSERT`
- ... which is where we are

The Next Step

- Adding more assertions

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions
  - SAFE to ASSERT

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions
  - SAFE to ASSERT
  - ASSERT to OPT

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions
  - SAFE to ASSERT
  - ASSERT to OPT
- Changing deployed assertion levels

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions
  - SAFE to ASSERT
  - ASSERT to OPT
- Changing deployed assertion levels
- Everyone will need to do this in 202x!

The Next Step

- Adding more assertions
  - `~bsl::string() { ASSERT(m_data[m_size] == 0); }`
  - Time ABI change
- Changing levels of assertions
  - SAFE to ASSERT
  - ASSERT to OPT
- Changing deployed assertion levels
- Everyone will need to do this in 202x!
  - Using language contracts when they come will be a case of adding new assertions to existing code.

Mis-Step #1

- Continuing violation handler (2008-2015)

Mis-Step #1

- Continuing violation handler (2008-2015)
  - Requires cooperation from main

Mis-Step #1

- Continuing violation handler (2008-2015)
  - Requires cooperation from main
  - Allows new bugs to go through unnoticed

Mis-Step #1

- Continuing violation handler (2008-2015)
  - Requires cooperation from main
  - Allows new bugs to go through unnoticed
  - At least 1 major Bloomberg (WP) bug was because of this

Mis-Step #1

- Continuing violation handler (2008-2015)
  - Requires cooperation from main
  - Allows new bugs to go through unnoticed
  - At least 1 major Bloomberg (WP) bug was because of this
  - Blanket continuation unsafe

Mis-Step #2

- Extra Smart violation handler (2016)

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts
  - Alternate logging

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts
  - Alternate logging
- Still unsuccessful

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts
  - Alternate logging
- Still unsuccessful
  - Requires even more cooperation from main

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts
  - Alternate logging
- Still unsuccessful
  - Requires even more cooperation from main
  - No way to indicate in code that a check is “new”

Mis-Step #2

- Extra Smart violation handler (2016)
  - Configuration to allow continuation
  - Tracking failure counts
  - Alternate logging
- Still unsuccessful
  - Requires even more cooperation from main
  - No way to indicate in code that a check is “new”
  - Rarely used, minimal progress

Step?

- `BSLS_REVIEW` (2018)

Step?

- `BSLS_REVIEW` (2018)
  - No explicit cooperation from main needed

Step?

- `BSLS_REVIEW` (2018)
  - No explicit cooperation from main needed
  - Contracts can be marked as “new” in code

Step?

- `BSLS_REVIEW` (2018)
  - No explicit cooperation from main needed
  - Contracts can be marked as “new” in code
  - Build-time controls to mark all assertions at a level as “new”

Step?

- `BSLS_REVIEW` (2018)
  - No explicit cooperation from main needed
  - Contracts can be marked as “new” in code
  - Build-time controls to mark all assertions at a level as “new”

BSLS_REVIEW overview

- Parallel structure to `BSLS_ASSERT`

BSLS_REVIEW overview

- Parallel structure to `BSLS_ASSERT`
- Separate violation handler, defaults to logging

BSLS_REVIEW overview

- Parallel structure to `BSLS_ASSERT`
- Separate violation handler, defaults to logging
- Lifecycle `BSLS_ASSERT_SAFE` to `BSLS_REVIEW` to `BSLS_ASSERT`

BSLS_REVIEW overview

- Parallel structure to `BSLS_ASSERT`
- Separate violation handler, defaults to logging
- Lifecycle `BSLS_ASSERT_SAFE` to `BSLS_REVIEW` to `BSLS_ASSERT`
- Alternately, <nothing> to `BSLS_REVIEW_?` to `BSLS_ASSERT_?`

BSLS_REVIEW

- Initially a copy of ASSERT

```
#define REVIEW_IMP(X) if (!(X)) {                                    \
    bb::assert_violation violation(__FILE__, __LINE__, #X);          \
    bb::Review::invoke_violation_handler(violation);                 \
}
```

BSLS_REVIEW

- Initially a copy of ASSERT
- Number of failures is important

```
#define REVIEW_IMP(X) if (!(X)) {                                          \
    static std::atomic<int> count;                                         \
    bb::review_violation violation(__FILE__, __LINE__, ++count, #X);\
    bb::Review::invoke_violation_handler(violation);                       \
}
```

BSLS_REVIEW

- Initially a copy of ASSERT
- Number of failures is important
- Default violation handler logs only

```
void Review::default_violation_handler(
                                const bb::review_violation &violation)
{
  // Log a message, with contents of violation
  // Log a stack trace
  // Return
}
```

BSLS_REVIEW

- Initially a copy of ASSERT
- Number of failures is important
- Default violation handler logs only
- With expeonential backoff

```
void Review::default_violation_handler(
                                const bb::review_violation &violation)
{
  int count = violation.count();
  if (0 == (count & (count-1))) {
    // Log a message, with contents of violation
    // Log a stack trace
  }
  // Return
}
```

BSLS_REVIEW Build time control

- Mutually exlusive

```
#if defined(REVIEW_LEVEL_NONE)   ? 1 : 0 \
  + defined(REVIEW_LEVEL_OPT)    ? 1 : 0 \
  + defined(REVIEW_LEVEL_REVIEW) ? 1 : 0 \
  + defined(REVIEW_LEVEL_SAFE)   ? 1 : 0 \
  > 1
#error Multiple REVIEW_LEVEL macros defined
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level

```
#if defined(ASSERT_LEVEL_NONE)
#define REVIEW_LEVEL_NONE
#elif defined(ASSERT_LEVEL_OPT)
#define REVIEW_LEVEL_OPT
#elif defined(ASSERT_LEVEL_ASSERT)
#define REVIEW_LEVEL_REVIEW
#elif defined(ASSERT_LEVEL_SAFE)
#define REVIEW_LEVEL_SAFE
#else
#define REVIEW_LEVEL_REVIEW
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level (In reality copies assert logic)

```
#if defined(ASSERT_LEVEL_NONE)
#define REVIEW_LEVEL_NONE
#elif defined(ASSERT_LEVEL_OPT)
#define REVIEW_LEVEL_OPT
#elif defined(ASSERT_LEVEL_ASSERT)
#define REVIEW_LEVEL_REVIEW
#elif defined(ASSERT_LEVEL_SAFE)
#define REVIEW_LEVEL_SAFE
#else
#define REVIEW_LEVEL_REVIEW
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
#if defined(REVIEW_LEVEL_NONE)
#define REVIEW_OPT(X)  REVIEW_DISABLED_IMP(X)
#define REVIEW(X)      REVIEW_DISABLED_IMP(X)
#define REVIEW_SAFE(X) REVIEW_DISABLED_IMP(X)
//..
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
//..
#elif defined(REVIEW_LEVEL_OPT)
#define REVIEW_OPT(X)  REVIEW_IMP(X)
#define REVIEW(X)      REVIEW_DISABLED_IMP(X)
#define REVIEW_SAFE(X) REVIEW_DISABLED_IMP(X)
//..
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
//..
#elif defined(REVIEW_LEVEL_REVIEW)
#define REVIEW_OPT(X)  REVIEW_IMP(X)
#define REVIEW(X)      REVIEW_IMP(X)
#define REVIEW_SAFE(X) REVIEW_DISABLED_IMP(X)
//..
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
//..
#elif defined(REVIEW_LEVEL_SAFE)
#define REVIEW_OPT(X)  REVIEW_IMP(X)
#define REVIEW(X)      REVIEW_IMP(X)
#define REVIEW_SAFE(X) REVIEW_IMP(X)
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
#if defined(REVIEW_LEVEL_OPT)     \
  defined(REVIEW_LEVEL_REVIEW) \
  defined(REVIEW_LEVEL_SAFE)
#define REVIEW_OPT(X) REVIEW_IMP(X)
#else
 // defined(REVIEW_LEVEL_NONE)
#define REVIEW_OPT(X) REVIEW_DISABLED_IMP(X)
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
#if defined(REVIEW_LEVEL_REVIEW) \
  defined(REVIEW_LEVEL_SAFE)
#define REVIEW(X) REVIEW_IMP(X)
#else
 // defined(REVIEW_LEVEL_OPT)
 // defined(REVIEW_LEVEL_NONE)
#define REVIEW(X) REVIEW_DISABLED_IMP(X)
#endif
```

BSLS_REVIEW Build time control

- Mutually exlusive
- Default to assert level
- Controls just like ASSERT

```
#if defined(REVIEW_LEVEL_SAFE)
#define REVIEW_SAFE(X) REVIEW_IMP(X)
#else
 // defined(REVIEW_LEVEL_OPT)
 // defined(REVIEW_LEVEL_REVIEW)
 // defined(REVIEW_LEVEL_NONE)
#define REVIEW_SAFE(X) REVIEW_DISABLED_IMP(X)
#endif
```

ASSERT and REVIEW interaction

- Changing build levels requires reviewing all asserts at the target level

ASSERT and REVIEW interaction

- Changing build levels requires reviewing all asserts at the target level
- `BSLS_ASSERT` again

```
#if defined(BSLS_ASSERT_LEVEL_ASSERT) \
  defined(BSLS_ASSERT_LEVEL_SAFE)
#define BSLS_ASSERT(X) ASSERT_IMP(X)
#else
#define BSLS_ASSERT(X)
#endif
```

ASSERT and REVIEW interaction

- Changing build levels requires reviewing all asserts at the target level
- `BSLS_ASSERT` again

```
#if defined(BSLS_ASSERT_LEVEL_ASSERT)  \
  defined(BSLS_ASSERT_LEVEL_SAFE)
#define BSLS_ASSERT(X) ASSERT_IMP(X)
#elif defined(BSLS_REVIEW_LEVEL_REVIEW) \
    defined(BSLS_REVIEW_LEVEL_SAFE)
#define BSLS_ASSERT(X) REVIEW_IMP(X)
#else
#define BSLS_ASSERT(X) ASSERT_DISABLED_IMP(X)
#endif
```

ASSERT and REVIEW interaction

- Changing build levels requires reviewing all asserts at the target level
- `BSLS_ASSERT` again
- Same for `BSLS_ASSERT_OPT` and `BSLS_ASSERT_SAFE`

```
#if defined(BSLS_ASSERT_LEVEL_OPT)     \
  defined(BSLS_ASSERT_LEVEL_ASSERT) \
  defined(BSLS_ASSERT_LEVEL_SAFE)
#define BSLS_ASSERT_OPT(X) ASSERT_IMP(X)
#elif defined(BSLS_REVIEW_LEVEL_OPT)   \
  defined(BSLS_REVIEW_LEVEL_REVIEW) \
  defined(BSLS_REVIEW_LEVEL_SAFE)
#define BSLS_ASSERT_OPT(X) REVIEW_IMP(X)
#else
#define BSLS_ASSERT_OPT(X) ASSERT_DISABLED_IMP(X)
#endif
```

ASSERT and REVIEW interaction

- Changing build levels requires reviewing all asserts at the target level
- `BSLS_ASSERT` again
- Same for `BSLS_ASSERT_OPT` and `BSLS_ASSERT_SAFE`

```
#if defined(BSLS_ASSERT_LEVEL_SAFE)
#define BSLS_ASSERT_SAFE(X) ASSERT_IMP(X)
#elif defined(BSLS_REVIEW_LEVEL_SAFE)
#define BSLS_ASSERT_SAFE(X) REVIEW_IMP(X)
#else
#define BSLS_ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
#endif
```

BSLS_REVIEW Takeaways

- So far a success

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed
  - No crashes have been introduced by these changes

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed
  - No crashes have been introduced by these changes
- Why?

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed
  - No crashes have been introduced by these changes
- Why?
  - Ability to make a check a review alongside existing asserts.

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed
  - No crashes have been introduced by these changes
- Why?
  - Ability to make a check a review alongside existing asserts.
  - Can conrol from code

BSLS_REVIEW Takeaways

- So far a success
  - Thousands of `BSLS_ASSERT_SAFE` instances have become `BSLS_REVIEW`
  - More than 90% are now `BSLS_ASSERT`
  - Dozens of reported bugs have been fixed/are being fixed
  - No crashes have been introduced by these changes
- Why?
  - Ability to make a check a review alongside existing asserts.
  - Can conrol from code
  - Can control at build time

What do you do if you can't prove a contract is being followed?

What do you do if you can't prove a contract is being followed?

Believe

We were promised performance

- Performance improvements come from the compiler *knowing* something will be true

We were promised performance

- Performance improvements come from the compiler *knowing* something will be true
- `[[noreturn]]` on `invoke_violation_handler` lets you safely trade the cost of checking for the benefit of assumption

We were promised performance

- Performance improvements come from the compiler *knowing* something will be true
- `[[noreturn]]` on `invoke_violation_handler` lets you safely trade the cost of checking for the benefit of assumption
- If you believe the contract is being followed, `__builtin_assume` can give you the benefit without the costt

We were promised performance

- Performance improvements come from the compiler *knowing* something will be true
- `[[noreturn]]` on `invoke_violation_handler` lets you safely trade the cost of checking for the benefit of assumption
- If you believe the contract is being followed, `__builtin_assume` can give you the benefit without the costt
- The risk is the strength of your belief

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros

```
#define BSLS_ASSERT_ASSUME(X) if (!(X)) { std::unreachable(); }
```

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros
- Lots of ways to impelement, different tradeoffs and portability

```
#define BSLS_ASSERT_ASSUME(X) if (!(X)) { std::unreachable(); }
#define BSLS_ASSERT_ASSUME(X) __builtin_assume(X)
#define BSLS_ASSERT_ASSUME(X) if (!(X)) { int *p = nullptr; *p = 17; }
```

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros
- Lots of ways to impelement, different tradeoffs and portability
- This almost made it to the standard

```
#define BSLS_ASSERT_ASSUME(X) [[ assert assume : X ]]
```

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros
- Lots of ways to impelement, different tradeoffs and portability
- This almost made it to the standard
- Coming to BDE with an extended `BSLS_ASSERT_LEVEL` scale

```
//..
#elif defined(ASSERT_LEVEL_ASSUME_OPT)
#define ASSERT_OPT(X)  ASSERT_ASSUME(X)
#define ASSERT(X)      ASSERT_DISABLED_IMP(X)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
//..
```

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros
- Lots of ways to impelement, different tradeoffs and portability
- This almost made it to the standard
- Coming to BDE with an extended `BSLS_ASSERT_LEVEL` scale

```
//..
#elif defined(ASSERT_LEVEL_ASSUME_OPT)
#define ASSERT_OPT(X)  ASSERT_ASSUME(X)
#define ASSERT(X)      ASSERT_ASSUME(X)
#define ASSERT_SAFE(X) ASSERT_DISABLED_IMP(X)
//..
```

BSLS_ASSERT_LEVEL_ASSUME

- Let's add another choice for mapping the `BSLS_ASSERT` macros
- Lots of ways to impelement, different tradeoffs and portability
- This almost made it to the standard
- Coming to BDE with an extended `BSLS_ASSERT_LEVEL` scale

```
//..
#elif defined(ASSERT_LEVEL_ASSUME_OPT)
#define ASSERT_OPT(X)  ASSERT_ASSUME(X)
#define ASSERT(X)      ASSERT_ASSUME(X)
#define ASSERT_SAFE(X) ASSERT_ASSUME(X)
//..
```

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues
  - Continuation was a global flag, and its very existence was contentious

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues
  - Continuation was a global flag, and its very existence was contentious
  - Assumption of any unchecked contracts

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues
  - Continuation was a global flag, and its very existence was contentious
  - Assumption of any unchecked contracts
  - Axiom was isomorphic to `__builtin_assume`

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues
  - Continuation was a global flag, and its very existence was contentious
  - Assumption of any unchecked contracts
  - Axiom was isomorphic to `__builtin_assume`
  - **`default`**/`audit`/`axiom` were both too simplistic and too complicated for many

What happened?

- Coming into Kona, contracts (in N4800) had a number of issues
  - Continuation was a global flag, and its very existence was contentious
  - Assumption of any unchecked contracts
  - Axiom was isomorphic to `__builtin_assume`
  - **`default`**/`audit`/`axiom` were both too simplistic and too complicated for many
  - Numerous edge case decisions had been made without publicizing clearly their reasoning

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems
- On Monday, July 15th in Cologne a number of options were presented to EWG

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems
- On Monday, July 15th in Cologne a number of options were presented to EWG
  - P1711, by Bjarne Stroustrup, proposed some small fixes

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems
- On Monday, July 15th in Cologne a number of options were presented to EWG
  - P1711, by Bjarne Stroustrup, proposed some small fixes
  - P1429, revised after Kona, proposed adding literal semantics

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems
- On Monday, July 15th in Cologne a number of options were presented to EWG
  - P1711, by Bjarne Stroustrup, proposed some small fixes
  - P1429, revised after Kona, proposed adding literal semantics
  - P1607 proposed two options - remove all but a nicer c assert, or remove all but add in literal semantics

What happened?

- Numerous papers in Kona and Cologne attempted to fix these problems
- On Monday, July 15th in Cologne a number of options were presented to EWG
  - P1711, by Bjarne Stroustrup, proposed some small fixes
  - P1429, revised after Kona, proposed adding literal semantics
  - P1607 proposed two options - remove all but a nicer c assert, or remove all but add in literal semantics
- P1607's two options were the only consensus reached that day by EWG

What happened?

- On Wednesday, July 17th in Cologne P1823 was proposed and accepted by a massive margin

What happened?

- On Wednesday, July 17th in Cologne P1823 was proposed and accepted by a massive margin
- On Saturday, July 20th, P1823 was ratified and SG21 was announced to pursue contracts again

What's the plan?

- SG21 is getting off the ground:

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830
    - Changes in syntax? Scope? Behaviors?

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830
    - Changes in syntax? Scope? Behaviors?
  - Review and vote on solutions based on satisfying use cases
    - Hopefullly no more "union of minimal solutions"

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830
    - Changes in syntax? Scope? Behaviors?
  - Review and vote on solutions based on satisfying use cases
    - Hopefullly no more "union of minimal solutions"
    - Hopefully no more "I don't need this so it can't happen"

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830
    - Changes in syntax? Scope? Behaviors?
  - Review and vote on solutions based on satisfying use cases
    - Hopefullly no more "union of minimal solutions"
    - Hopefully no more "I don't need this so it can't happen"
- Land revised contracts in standard

What's the plan?

- SG21 is getting off the ground:
  - Very active reflector
  - One telecon so far
  - First official meeting will be in Belfast
- Rough sketch of plan:
  - Gather use cases publicly (Done!)
  - Poll on prioritization of use cases (In progress!)
  - Gather proposed solutions (Future)
    - Refinements on P1607, N4830
    - Changes in syntax? Scope? Behaviors?
  - Review and vote on solutions based on satisfying use cases
    - Hopefullly no more "union of minimal solutions"
    - Hopefully no more "I don't need this so it can't happen"
- Land revised contracts in standard (WG21 SG21 2021!)

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form “As an X in order to Y I need to Z”.

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form “As an X in order to Y I need to Z”.
- This hopefully be the first wg21 paper published “by” SG21.

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form “As an X in order to Y I need to Z”.
- This hopefully be the first wg21 paper published “by” SG21.
- 29 different classes of users with 196 different use cases

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form “As an X in order to Y I need to Z”.
- This hopefully be the first wg21 paper published “by” SG21.
- 29 different classes of users with 196 different use cases
- Use cases range from very general to very specific:

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form "As an X in order to Y I need to Z".
- This hopefully be the first wg21 paper published "by" SG21.
- 29 different classes of users with 196 different use cases
- Use cases range from very general to very specific:
  - As a Developer, in order to Have readable annotations, I want to Have annotations with a succinct and elegant syntax

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form "As an X in order to Y I need to Z".
- This hopefully be the first wg21 paper published "by" SG21.
- 29 different classes of users with 196 different use cases
- Use cases range from very general to very specific:
  - As a Developer, in order to Have readable annotations, I want to Have annotations with a succinct and elegant syntax
  - As a C++ API Developer, in order to Maintain a class hierarchy, I want to Ensure overriding methods have same or wider preconditions

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form "As an X in order to Y I need to Z".
- This hopefully be the first wg21 paper published "by" SG21.
- 29 different classes of users with 196 different use cases
- Use cases range from very general to very specific:
  - As a Developer, in order to Have readable annotations, I want to Have annotations with a succinct and elegant syntax
  - As a C++ API Developer, in order to Maintain a class hierarchy, I want to Ensure overriding methods have same or wider preconditions
- Some require vastly more than was possible before:

Use Cases

- Use cases were gathered from all SG21 participants and edited for clarity
- All of the form "As an X in order to Y I need to Z".
- This hopefully be the first wg21 paper published "by" SG21.
- 29 different classes of users with 196 different use cases
- Use cases range from very general to very specific:
  - As a Developer, in order to Have readable annotations, I want to Have annotations with a succinct and elegant syntax
  - As a C++ API Developer, in order to Maintain a class hierarchy, I want to Ensure overriding methods have same or wider preconditions
- Some require vastly more than was possible before:
  - As a C++ API Developer In Order to Enforce contracts in async code I want to Express contracts on callbacks such as std::function, function pointers, or references to functions, lambdas, or function objects

Prioritization

- SG21 members have been asked to rate each use case on behalf of whatever users they feel they best represent:
  - Not important
  - Nice to have
  - Must have

Prioritization

- SG21 members have been asked to rate each use case on behalf of whatever users they feel they best represent:
  - Not important
  - Nice to have
  - Must have
- Expect these results to be ready to analyze and discuss by Belfast.

Conclusion

- Bloomberg's `BSLS_ASSERT` and `BSLS_REVIEW` provide a rich set of contract enforcement utility. Grab the open source BDE to play with it today
- The needs of that facility will hopefully be met by language level contracts in the future, SG21 is working hard to see that happen