Progress toward Contract Support in C++17

Nathan Myers
Bloomberg LP

May 12, 2016
ncm@cantrip.org

CppNow 2016

Background

Contracts are not new:

```
void copy(char const* from, size_t n, char* to)
```

Precondition: `from != 0 && to != 0`
Postcondition: `memcmp(from, to, n) == 0`

.

Background

Contracts are not new:

```
void copy(char const* from, size_t n, char* to)
```

Precondition: `from != 0 && to != 0 || n == 0`
Postcondition: `memcmp(from, to, n) == 0`

Background

Contracts are not new:

```
void copy(char const* from, size_t n, char* to)
```

Precondition: `from != 0 && to != 0 || n == 0`
Postcondition: `memcmp(from, to, n) == 0`

- Users get angry without this addition
- Formal contracts need more care and testing than we might be used to doing

Background

Contracts are not new:

```
void copy(char const* from, size_t n, char* to)
```

Precondition: `from != 0 && to != 0 || n == 0`
Postcondition: `memcmp(from, to, n) == 0`

- Documentation & comments only
- Limited rigor, sloppy design, inconsistencies
  - `memcmp` allows null `from`, `to` when n == 0
  - `memcpy` has undefined behavior if any argument is zero

Background

Contracts are not new:

```
void copy(char const* from, size_t n, char* to)
```

Precondition: `from != 0 && to != 0 || n == 0`
Postcondition: `memcmp(from, to, n) == 0`

Another example:

```
bool
binary_search(int const* b, int const* e, int v)
```

Precondition: `b == e ||`
`less(b, e) && is_partitioned(b, e, v)`
Returns: `find(b, e, v) != e`

Background

Wide contract:

No preconditions

Background

Wide contract:

No preconditions

```
std::vector<T>::size()
std::vector<T>::push_back(T&&)
```

Background

Wide contract:

No preconditions

```
std::vector<T>::size()
std::vector<T>::push_back(T&&)
```

Widen a contract:

```
copy(char* from, size_t n, char* to)
```

Requires: `from != 0 && to != 0`

Background

Wide contract:

No preconditions

```
std::vector<T>::size()
std::vector<T>::push_back(T&&)
```

Widen a contract:

```
copy(char* from, size_t n, char* to)
```

Requires: `from != 0 && to != 0 || n == 0`

Background

Wide contract:

No preconditions

```
std::vector<T>::size()
std::vector<T>::push_back(T&&)
```

Widen a contract:

```
copy(char* from, size_t n, char* to)
```

Requires: `from != 0 && to != 0 || n == 0`

Narrow a contract: . . .

Background

Wide contract:

No preconditions

```
std::vector<T>::size()
std::vector<T>::push_back(T&&)
```

Widen a contract:

```
copy(char* from, size_t n, char* to)
```

Requires: `from != 0 && to != 0`

Narrow a contract: . . .

Background

Undefined Behavior

Background

Undefined Behavior

```
char from[10], to[10];
memcpy(to, from, 10);  // boom -- uninitialized
```

Background

Undefined Behavior

```
char from[10], to[10];
memcpy(to, from, 10);  // boom
```

“Soft” Undefined Behavior

```
copy(0, 0, 0);          // boom?
```

Background

Undefined Behavior

```
char from[10], to[10];
memcpy(to, from, 10);  // boom
```

"Soft" Undefined Behavior

```
copy(0, 0, 0);          // boom?
```

- *User* must assume the implementation does something undefined

  It might! Or, maybe we got here because of previous UB

Background

Undefined Behavior

```
char from[10], to[10];
memcpy(to, from, 10);  // boom!
```

"Soft" Undefined Behavior

```
copy(0, 0, 0);          // boom?
```

- *User* must assume the implementation does something undefined

  It might! Or, maybe we got here because of previous UB

- *Compiler* cannot assume the call results in UB

  Implementation might check

Background

C-style assert

```
#include <cassert>
. . .
assert(p != 0);
```

Background

C-style assert

```
#include <cassert>
. . .
assert(p != 0);
```

- Optimizer can't see it
  - `#define assert(x)`

Background

C-style assert

```
#include <cassert>
. . .
assert(p != 0);
```

- Optimizer can't see it
  - `#define assert(x)`
- Optimizer not allowed to act on it

Background

C-style assert

```
#include <cassert>
. . .
assert(p != 0);
```

- Optimizer can't see it
  - `#define assert(x)`
- Optimizer not allowed to act on it
  - `#define assert(x)  __builtin_assume(x)`

  … unless specifically permitted

Goals

- Static Correctness
  - Build can fail on misuse detected by compiler
  - `constexpr` arguments might be traced through inlines/templates, maybe several levels deep
  - Return-value and postcondition properties can be used to check subsequent preconditions: "stitching"
  - 3rd party tools might do deeper analysis than the compiler can afford to do

Goals

- Static Correctness
- Runtime Correctness
  - Programmed runtime response to detected misuse
  - E.g., call a handler: log details, maybe save state, maybe clean up
  - Beta release might phone home, send log on restart

Goals

- Static Correctness
- Runtime Correctness
- Better Security?
  - Error handler set at link time
  - Runtime postconditions – buffer zeroed

Goals

- Static Correctness
- Runtime Correctness
- Better Security?
- Better Performance
  - Compiler needs permission to use assertions in optimization
  - Code analysis is NP, assertions may be used as *oracles*
  - Assertions can reveal what even whole-program analysis ("link-time optimization") cannot
  - Performance gains reduce pressure to (unwisely) put misuse handling in the interface

Goals

- Better Static Correctness
- Better Runtime Correctness
- Better Security?
- Better Performance
- Better Testing
  - Test cases exercise assertions in addition to checking output
  - Assertions cross-check library dependencies, up and down layers
  - For some, a poor man's substitute for unit tests (!)

Goals

- Better Static Correctness
- Better Runtime Correctness
- Better Security?
- Better Performance
- Better Testing
- Better Bug Reports / Better Library Experience
  - Debug version logs both internal failures and misuse
  - Checks prevent bug reports for usage mistakes
  - Maintainers less distracted by spurious bug reports

Complications

- Fundamental problems

Complications: Fundamental Problems

Incomplete Capture

Complications: Fundamental Problems

Incomplete Capture

- Inexpressible:

  `void operator delete(void* p)`

  Pre: p obtained from op new, not since deleted (or 0)

  Post: p available to return from op new again, p undefined

Complications: Fundamental Problems

Incomplete Capture

- Inexpressible:

  `void operator delete(void* p)`

  Pre: `p` obtained from op `new`, not since deleted (or 0)

  Post: `p` available to return from op `new` again, `p` undefined

- Impractical:

  `void sort(b, e)` `// O(n log n)`

  Pre: `b, e` are from same container, `b < e`

  Post: `is_sorted(b, e)` `// O(n)`

  Post: `is_permutation(b, e, . . .` `// O(n²)`

Complications: Fundamental Problems

Incomplete Capture

- Inexpressible:

  `void operator delete(void* p)`

  Pre: `p` obtained from op `new`, not since deleted (or 0)

  Post: `p` available to return from op `new` again, `p` undefined

- Impractical:

  `void sort(b, e)` `// O(n log n)`

  Pre: `b, e` are from same container, `b < e`

  Post: `is_sorted(b, e)` `// O(n)`

  Post: `is_permutation(b, e, . . .` `// O(n²)`

  (Permutation of what? Saving copies of `b, e` *not enough)*

Complications: Fundamental Problems

- Incomplete Capture
- Cluttered Interface
  - Contract junk can be much longer than the declaration
  - Too long, yet incomplete
  - Expressions only – i.e. “functional”, not very expressive :-)

Complications: Fundamental Problems

- Incomplete Capture
- Cluttered Interface
- Aliasing
  - Many ways to get "directly" to a function
    - Virtual inheritance
    - Function pointers
    - (Partial) specialization
    - Function template overload

Complications: Fundamental Problems

- Incomplete Capture
- Cluttered Interface
- Aliasing
  - Many ways to get "directly" to a function
    - Virtual inheritance
    - Function pointers
    - (Partial) specialization
    - Function template overload
  - Potentially different contract terms
    - Narrowing, trivially
    - Widening, in certain circumstances (!)

Complications

- Fundamental problems
- Runtime Checking

Complications: Runtime Checking

- Catches mistakes static checking cannot, but…
- Adds run-time cost
  - No upper limit to potential cost
  - But very cheap checks can catch common mistakes
- Needs a better response to violations than `abort()`, i.e. user-specified
- Thorough checking often violates specified complexity
- … motivating "audit-mode" builds, & not => implies build modes
- Linking mixed-mode builds suggests ODR-"ish" questions – which instantiation of a template / inline do you get, the version with checking, or without?

Complications

- Fundamental problems
- Runtime Checking Levels
- Optimization

Complications: Optimization

- Check expressions can be useful to the optimizer
  - if violation handler can't return, forward-propagate implications
  - if all runtime checking is turned off, backward-propagate too
  - check expression itself has implications useful to the optimizer

```
[[assert: *p == '#']]  // implicitly, assume(p)
```

  - but too many overwhelms the optimizer, and it gives up!
- Check-expression implication can elide later check expressions
- Checks may be marked never to execute: "axiom"
- Axioms can call unimplemented / unimplementable functions, just for their declared pre / postconditions or built-in implications

Complications

- Fundamental problems
- Runtime Checking Levels
- Optimization
- Build modes

Complications: Build Modes

- No one-size-fits-all
- Runtime checking levels, *“audit” / (default) / “never”*
- Runtime violations: *can / cannot* resume, if handler returns
- Runtime violation handler: *default “abort” / link-time specified*
- Optimizer *allowed / not allowed* to treat non-runtime checks as *oracles*
- Maybe, check *in caller / in callee,* enabling retrofitting old libraries

Process

- Players
- Timeline

Process: Players

- Bloomberg: runtime checking, improved testing
- Academia, Coverity: static checking, program correctness
- Microsoft: security, correctness
- Google, Nvidia, Facebook: optimization

Process: Timeline

- 2010: Proposal based on Bloomberg runtime-checking macros

  . . . *(things happen)*

- 2013: Library proposal approved in subcommittees LEWG, LWG
- 2014: Library proposal ejected by full committee: "Macros! ODR!"
- 2015: Bloomberg/Microsoft/other competing/conflicting core proposals
  - Run-time/compile-time vs. compile-time only
  - Checks in declarations vs. in function bodies

    => Committee requests joint proposal
- 2016: Joint design proposed, welcomed; more detailed proposal in Oulu
- Future: TS, then in standard 2019/2020, integrate to std lib spec after
- C++17 implementations will implement the TS, use in std lib; you can too

Details

- Syntax
- Transition
- Runtime Postconditions
- Runtime Checking Control
- Static Declarations
- Optimization

Details: syntax

- *attributes* (with certain core-grammar changes), by example:

```
template <typename It, typename Cmp>
void sort(It b, It e, Cmp cmp)
    [[pre axiom: reaches(b, e)]]   // notional! Not expressible
  - [[pre audit: [=]() {           // check predicate behavior
        for (It p = b; p != e; ++p)
            for (It q = p; q != e; ++q)
                if cmp(*p, *q) == cmp(*q, *p) return false;
        return true; }() ]]
    [[post: is_sorted(b, e)]]
{
    It tb = b, te = e;
    . . .
        [[assert: *tb < *te]];
    . . .
    [[assert: is_permutation(b, e, [what?])]];
}
```

- Not nailed down yet: commas? *attribute-token* order? names? return value?

Details: Transition

- Newly instrumenting any significant library breaks all programs that use it, until fixed

Details: Transition

- Newly instrumenting any significant library breaks all programs that use it, until fixed
  - Need transition support

Details: Transition

- Newly instrumenting any significant library breaks all programs that use it, until fixed
  - Need transition support
  - Need C++14 compilers to permit new contract-attribute syntax, ignore new attributes, but still warn about misspellings of known attributes, e.g.:

    ```
    -Wno-unknown-attribute=pre,post,assert,audit,axiom
    ```

Details: Transition

- Newly instrumenting any significant library breaks all programs that use it, until fixed
  - Need transition support
  - Need C++14 compilers to permit new contract-attribute syntax, ignore new attributes, but still warn about misspellings in known attributes, e.g.:

    ```
    -Wno-unknown-attribute=pre,post,assert,audit,axiom
    ```
  - Maintaining separate library versions (instrumented and not-instrumented) for use with different compilers is unpleasant (and involves many, many macros)

Details: Standard Library

The C++ Standard Library *counts as a significant library*

- Expect C++17 implementations to have annotated their respective std libs
- Expect all your programs to break: you will need to rely on transition aids until your programs and libraries are fixed

Details: Runtime Postconditions

Runtime postconditions have a *wee* problem:

Pass-by-value arguments may appear in the postcondition

- Static checks want the values the arguments had on entry
  - Entry, exit, and return values all "exist", statically
    (Might need a way to mention both)
- Runtime checks want . . . what, exactly?

```
template <typename It, typename Cmp>
void sort(It b, It e, Cmp cmp)
    [[post: is_sorted(b, e)]]       // did sort change b, e?
    [[pre axiom: reaches(b, e)]];   // notional
```

Details: Runtime Postconditions

Runtime postconditions have a *wee* problem

- Value arguments may appear in the postcondition
  - Static checks want the value on entry
    - Values before and values after both "exist", statically
    - Might need a way to mention both
  - Runtime checks want . . . what, exactly?
- Mutating a pass-by-value argument in the function body, if it was mentioned in a postcondition, is "ill-formed"
  - . . . but they are not `const` (const values are mutable too); anyway,
  - . . . lookup must be the same with or without the postcondition
  - If the compiler sees a violation, it must report an error – but if it cannot see a violation, "no diagnostic required"

Details: Runtime Checking Control

- Controlling whether to check contract annotations is complicated
- Remember: ***"In a correct program, all requirements are satisfied"***
- Compiler command line flags

```
--contract-runtime-level=[audit|none]
--contract-violation-handler=app::log_and_quit
--contract-violation-resume
```

- Sanitizer violations could also call the contract violation handler

```
--contract-sanitize                    # maybe?
```

Details: Static Declarations

- [[pre: …]], [[post:]] on function declarations
- All declarations of a given interface must have *identical* contract specifiers
  - Very restrictive (cf. aliasing), might be relaxed later
  - Must be "ODR-identical" (cf. inline-function definition rules)
- Tools can check individual calls, but also match postconditions on one call with preconditions on subsequent calls: "stitching"
- Declared, not-defined functions would be usable in axioms just for their pre- and postconditions
- Standard library can declare axioms with distinguished names known to the compiler, with meaning not necessarily expressible:

  `reaches(b, e)`, `null_terminated(s)`

Details: Optimization

- Optimization implications are very complicated
- Compiler needs permission to use contract annotations to optimize

  `--contract-assume=[axiom|audit|all|none]`”?
- Implications forward and backward? (Yes.)
- Can annotations pessimize code? (Yes.)

Details: Optimization

- Optimization implications are very, *very* complicated
- Interacts with `--contract-runtime-level`:

  Implications of checks evaluated at runtime propagate forward only
- If you *use* (dereference) a pointer in any contract annotation:
  - Does the pointer's implied non-null-ness propagate? (Yes.)
  - Forward *and* backward? (Yes.)
  - Potentially eliding subsequent runtime checks? (Yes.)
- More general annotations should follow more specific ones; otherwise, a smart enough optimizer might elide them

  Optimizers tend to get smarter

Immediate

- Better hacking with macros
- Hack into Clang, Gcc

Immediate: macros

- Can mostly implement runtime version with macros

```
#ifdef CHECK_CONTRACTS
#define contract_assert(x) ((x) ? \
    (void)0 : handler(__FILE__, __LINE__))
#else
#define contract_assert(x) __builtin_assume(x)
#endif
```

- Hijack standard `assert()`? UB, but implementation can define anything not otherwise defined.
- ODR violations – implementations get a free pass, users don't

Immediate: Hack Clang, Gcc

- Most infrastructure already in place
- Needs minor extensions to attribute-syntax processing
  - `:...]]` => `(...)]]`
  - *attribute-token attribute-token => attribute-token-pair*
- Patch C++14 compilers:

  `-Wno-unknown-attribute=pre,post,assert,audit,axiom`
- Tap into attribute handling, expression parsing, __builtin_assume
- Add `--contract-this`, `--contract-that`

Resources

- Walter Brown, "Proposing Contract Attributes" (w/comprehensive older references)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4435.pdf

- Bloomberg proposal motivating runtime support and providing standardese for function body assertions and run-time and compile-time semantics:

  "Language Support for Runtime Contract Validation"

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4378.pdf

  "FAQ about N4378, Language Support for Contract Assertions"

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4379.pdf

Resources

- J. Daniel Garcia, "Three interesting questions about contracts"

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0166r0.pdf
- Lakos, Meredith & Myers, "Contract Assert Support Merged Proposal" (an attempt)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0246r0.pdf
- Myers, "Criteria for Contract Support"

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0247r0.pdf
- Dos Reis et al, "Simple Contracts for C++"

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0287r0.pdf