C++ Contracts

a Meaningfully Viable Product

Andrei Zissu

C++ Contracts – a Meaningfully Viable Product

Andrei Zissu

Morphisec – preventive cyber security

- **WG21 member for the last 3 years**
- **Active in SG21**
  - **Not one of the main authors of the Contracts proposal (P2900)**

What This Talk Is About

- A moderately deep dive into contracts in C++26
- With some detail on considerations, controversies and paths not taken
- I'll strive to make the terminology easily accessible
  - Not always stick to the official terms
- I'll try not to focus on theory
  - But will sometimes present the important parts
  - Including relevant P2900 guiding principles, present in proper context
- I will be borrowing heavily from P2900 (the C++ contracts proposal, previously known as the contracts MVP) and P2899 (the rationale paper)
  - Occasionally other papers will also be mentioned and possibly borrowed from
- Not nearly enough time to cover everything I would have wanted to

Some quick terminology

Precondition Specifiers

```cpp
auto div(auto x, auto y)
    pre(y != 0)
{
    return x/y;
}

int main()
{
    return div(1, 0);
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function div<int, int> at /app/example.cpp:2: y != 0
  [assertion_kind: pre, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

- **“Precondition” is a plain language term, while “precondition specifier” refers to a syntactic construct.**

Postcondition Specifiers

```cpp
auto plus(const unsigned x, unsigned y)
    post(r: r >= x)
{
    return x - y;
}

int main()
{
    return plus(1, 1);
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function plus at /app/example.cpp:2: r >= x
  [assertion_kind: post, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

- The term “postcondition” is used in everyday language, whereas “postcondition specifier” denotes a specific syntactic element.
- *r* is directly naming the function return value – we’ll cover this later
- Also keep the const parameter in mind for later…

Function Contract Specifiers

- The collective term for all pre and post specifiers (for preconditions and postconditions)
- Sometimes referred to in short just as “Contract Specifiers”
- I will often refer to them simply as pre and post

Assertion Statements (a.k.a contract_assert)

```cpp
1  auto plus(unsigned x, unsigned y)
2  {
3      auto ret = x - y;
4      contract_assert(ret >= x);
5      return ret;
6  }
7
8  int main()
9  {
10     return plus(1, 1);
11 }
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function plus at /app/example.cpp:4: ret >= x
  [assertion_kind: assert, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

Contract Assertions

- The collective term for the syntactic construct specified by function contract specifiers (pre and post) and assertion statements.
- Yes, this includes pre and post, while contract_assert is referred to only as “assertion statement”.
- For convenience I will be referring to them mostly as “CA” – short for “contract assertion” (which is a bit of a mouthful)

Contract Predicates

- The C++ code inside a contract assertion
- Must be contextually convertible to bool
- We can usually mix and match the terms “contract assert” and “contract predicate”
  - But occasionally they do have important distinguishing differences
    - Like *ignore* semantics

Some Bikeshedding History
(or whatever you choose to name it)

Contract Assertions

- Started off as CCA (Contract-Checking Annotation) - renamed since some committee members didn’t think the term “annotations” is appropriate. Still being informally used (myself included).
- Also proposed: CAA (Contract-Assertion Annotation) - didn’t catch on
- I proposed "contract clauses”

contract_assert

- Other languages use “assert”.
- But that would conflict with the existing assert macro.
- So pending a future solution, we needed another name.
- Over 40 alternatives were collected and explored in P2961R2 - *A natural syntax for Contracts*

- **contractassert**
- **mustexpr**
- **dyn_assert**
- **musthold**
- **asrtexpr**
- **stdassert**
- **truexpr**
- **co_assert**
- **ccassert (this one is mine - disqualified as “too clever for most users”)**
- **contract_assert**
- **std_assert**
- **dyn_check**
- **mustbetrue**
- **assertexpr**
- **assertion_check**
- **cppassert**
- **dynamic_assert**
- **cca_assert**
- **assrt**
- **runtime_assert**
- **_Assert**
- **xpct**
- **assert_check**
- **Assert2**

- cpp_assert
- affirm
- __assert
- assess
- insist
- asrt
- cassert
- aver
- posit
- enforce
- audit
- claim
- ass
- must
- confirm
- assertion
- ensure
- chk
- verify
- expect
- check

- Finalists - contract_assert and assertexpr,
- contract_assert won the day.
- I voted against assertexpr as it resembles constexpr, which is used only in declarations. Given that contract_assert is currently a statement and not an expression (see later) I’m particularly glad we chose this name.

Syntactic restrictions, and how they came to be

Multiple Declarations
– because we love IFNDR!

- (Function) contract specifiers (pre/post) must always be declared on function first-declarations
  - First declaration (used but not defined by the standard – until P2900 that is) - “a function from which no other declaration is reachable” - i.e. 1st declaration found by compiler in current TU (a.k.a .cpp file)
    - Otherwise it’s a “redeclaration” - that includes function body when defined separately from first declaration
      - Contracts repetition in function redeclarations is allowed (not mandatory, since compilers don’t really need them)
        - Mainly for user friendliness and to avoid further include hell

Multiple Declarations
– because we love IFNDR!

- Ill-formed if repeated contract specifiers differ from reachable first declaration
  - Equivalence is determined by token equality (same as ODR) with renaming allowed for for function/template parameters and returned value
    - As long as the contract specifications do refer to the same entities
- Redeclarations in same TU cannot contain lambdas - would automatically render them “not the same” due to different type for each lambda

Multiple Declarations
– because we love IFNDR!

- **Well-formed renaming:**

```
struct C
{
    double div(double x)
        pre(x != 0);
};

double C::div(double y)
    pre(y != 0)
{
    return y;
}

int main()
{
    return C{}.div(1);
}
```

Multiple Declarations – because we love IFNDR!

- Ill-formed renaming:

```cpp
struct C
{
    double div(double x, double y)
        pre(y != 0);
};

double C::div(double x, double y)
    pre(x != 0)
{
    return x/y;
}

int main()
{
    return C{}.div(1, 1);
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A Wrap lines Select all

```
<source>:8:11: error: mismatched contract condition in declaration
    8 |     pre(x != 0)
      |           ~~^~~~
<source>:4:15: note: previous contract here
    4 |         pre(y != 0);
      |               ~~^~~~
Compiler returned: 1
```

Semantics

Constification
– our favorite bowel non-movement

- **officially called “implicit const-ness”**
- **often auto-corrected to “constipation** 
- **Main motivation:**
  - **catch bugs like “assert(my_map[“universal_answer”] == 42)”**
  - **avoid contracts producing side effects, whether inadvertently or intentionally**

> **Principle 6: No Destructive Side Effects**
>
> Contract assertions whose predicates, when evaluated, could affect the correctness of the program should not be supported.

  - not entirely clear cut - e.g. can the mere extra clock cycles affect program correctness?

Constification
– our favorite bowel non-movement

- **Effect:**
  - **Entities referenced by CAs (external to the CA itself) are treated similarly to data members in const member functions - with decltype still reporting the original constness**
  - **implicit const is shallow**
    - **objects pointed to by pointers are not constified**
    - **objects referred to by reference are constified**
  - **globals originally not included in constification - changed later to improve teachability**

```
int global = 0;

void f(int x, int y, char *p, int& ref)
  pre((x = 0) == 0)             // error: assignment to const lvalue
  pre((*p = 5))                 // OK
  pre((ref = 5))                // error: assignment to const lvalue
  pre((global = 2))             // error: assignment to const lvalue
{
  int* gp = &global;
  contract_assert((gp = nullptr)); // error: assignment to const lvalue
  contract_assert((*gp = 6));      // OK
}
```

Constification
– our favorite bowel non-movement

- controversial opt out - const_cast
  - must be used with utmost care, as the outcome is UB if the original object is const

```
int g(int i, const int j)
  pre(++const_cast<int&>(i))     // OK (but discouraged)
  pre(++const_cast<int&>(j))     // undefined behavior
  post(++const_cast<int&>(i))    // OK (but discouraged)
  post(++const_cast<int&>(j))    // undefined behavior
{
  int k = 0;
  const int l = 1;
  contract_assert(++const_cast<int&>(k));   // OK (but discouraged)
  contract_assert(++const_cast<int&>(l));   // undefined behavior
}
```

Constification
– our favorite bowel non-movement

- **Objections:**
  - **modifies overload resolution, therefore contracts may actually be evaluating different code**

```
struct X {};
bool p(X&) { return true; }
bool p(const X&) { return false; }

void my_assert(bool b) { if (!b) std::terminate(); }

void f(X x1)
  pre(p(x1))           // fails
{
  my_assert(p(x1));  // passes

  X x2;
  contract_assert(p(x2)); // fails
  my_assert(p(x2));       // passes
}
```

Constification
– our favorite bowel non-movement

- Objections:
  - modifies overload resolution, therefore contracts may actually be evaluating different code
    - counter - such overloads are bugs on their own
      - counter counter - contracts must be useful with real world code, not educate programmers
        - counter counter counter – on the contrary, contracts should encourage correct programming practices
    - counter - side effects in contracts create a different program – rendering such contracts potentially meaningless/harmful
    - counter - may help discover latent side effects in existing assert statements

Constification
– our favorite bowel non-movement

- Objections:
  - shallow const
    - constification is incomplete and inconsistent
    - in particular, pointers and references are inconsistent with each other
    - counter - pointer/reference inconsistency is an existing language feature
    - counter - automatic deep const would be very complex to specify and implement

Constification
– our favorite bowel non-movement

- **Objections:**
  - **constification blocks some common non-const usages, such as logging and std::map::operator[]**
    - **counter - std::map::operator[] is bad, std::map::at() should be used instead**
      - **no such easy solution with things like logging though**
    - **counter - const_cast as a controversial escape hatch**
    - **other proposed escape hatches:**
      - **mutable (applied to CA),**
      - **operator noconst (applied to expression in CA):**

```
// Legacy API, check_valid doesn't modify x but isn't const-qualified
struct X;
bool check_valid(X& x);

void f(X& x)
  pre(check_valid(x)); // Ill-formed due to implicit const on x

void g(X& x)
  pre(noconst(x).is_valid()); // OK: noconst(x) treats x as non-const for this expr
```

Postconditions

Postconditions- Referring to the Result Object – what is a name?

- Post may specify a name for the return value, valid only within the post itself
- That name will capture the result object "on the fly"
  - Even when the function returns an unnamed temporary - no other way in the language to do this for all objects
    - Although it is feasible for user-provided constructors, but only where those are available

Postconditions - Referring to the Result Object – what is a name?

- **Similar to references but not really one, also compared to structured bindings**
  - **Main difference is that decltype doesn't see it as a reference**

```cpp
#include <type_traits>

int f() post(r: std::is_reference_v<decltype(r)>)
{
    int x;
    int& ref = x;
    static_assert(std::is_reference_v<decltype(ref)>);
    return 42;
}

int main()
{
    f();
    return 0;
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

Wrap lines   Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
   contract violation in function f at /app/example.cpp:3: std::is_reference_v<decltype(r)>
   [assertion_kind: post, semantic: enforce, mode: predicate_false, terminating: yes]
   terminate called without an active exception
   Program terminated with signal: SIGSEGV
```

Postconditions- Referring to the Result Object – what is a name?

```
struct S {
  S();
  S(const S&) = delete; // noncopyable, nonmovable
  int i = 0;
  bool foo() const;
};

const S f()
  post(r: const_cast<S&>(r).i = 1)   // OK (but discouraged)
{
  return S{};
}

const S y = f();        // well-defined behavior
bool b = f().foo();   // well-defined behavior
```

- **const_cast, although discouraged, is actually safe in this case - result object being captured on the fly is not yet assigned to the call site target variable, therefore it cannot yet be const**

Referring to Parameters in Postconditions – wait, isn't const meaningless when passing by value?

- **TLDR: by-value function parameters odr-used (a.k.a consumed) in a post CA - must be const, in all the function declarations**

```
void f(int i) post ( i != 0 );            // error: i must be const.

void g(const int i) post ( i != 0 );
void g(int i) {}                          // error: missing const for i in definition

void h(const int i) post (i != 0);
void h(const int i) {}
void h(int i);                            // error: missing const for i in redeclaration
```

Referring to Parameters in Postconditions – wait, isn't const meaningless when passing by value?

- Motivation (also in P2521 - 3.13):
  - CAs refer to parameters directly (not to copies thereof)
  - post CAs are intended to work on initial parameter values
  - contracts impossible to reason about if parameters may mutate
- const must be consistently applied in all the function declarations
- with future contract capture clauses (not in C++26) function parameters won't have to be const – since intent will be explicit

Evaluation and Contract-Violation Handling

Point of Evaluation – who said it's a single point?

function call

- init params
  - pre
    - eval func body
      - contract_assert (when reached by control flow)
    - init result object (this means actual caller context object for non-trivially copyable returned types)
    - destruct local vars
  - post
- destruct params → bind result object

- pre/post see function parameters (or possibly their register copies) but not local variables
- result object is initialized before post (thus post can see it), but not yet considered bound
  - caller context const (if present) doesn't apply yet (meaning const_cast of result object in post is safe from UB)
- post is only evaluated on normal exit
  - evaluation when exiting due to exception may be possible in the future
- post is evaluated only after local variables destruction - since those might influence what post is checking

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- checking semantics
  - observe
  - terminating semantics
    - enforce
    - quick_enforce (was “Louis semantic”, prior to yet another bikeshedding procedure)
- ignore semantic
  - unlike C assert, it odr-uses and must be well-formed (which prevents code rot)

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- upon violation
    - at run time
        - observe
            - invoke violation handler
            - continue execution (upon normal return)
        - enforce
            - invoke violation handler
            - terminate (upon normal return)
        - quick_enforce
            - terminate immediately
    - at compile time
        - observe – issue diagnostic (a.k.a warning)
        - enforce and quick_enforce – make program ill-formed
    - ignore – doing nothing in both cases (other than odr-use)

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- selection of contract semantics
  - implementation-defined - because different implementations/platforms have different needs
    - no specification of when this happens
      - any build stage
      - run time
        - allowing dynamic contracts configuration
    - build modes (e.g. debug and release) are no longer required for this
      - but of course are still allowed
  - different semantics may be selected for different CAs in the same TU
    - or even for the same CA at different evaluations!

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- compliant implementations may offer any non-empty subset of the 4 semantics
  - meaning that ignore semantic alone is compliant, which entails:
    - enforcing CAs being well-formed
    - odr-use
      - side effects possible, e.g. initializing static data members of class templates
- chosen semantic explicitly undetectable from code at compile time
  - to avoid contracts changing observed behavior

Principle 3: Concepts Do Not See Contracts

The mere presence of a contract assertion on a function or in a block of code should not change the satisfiability of a concept, the result of overload resolution and SFINAE, the branch selected by `if constexpr`, or the value returned by the `noexcept` operator.

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- any CA may be evaluated 0…N times
  - not necessarily with the same semantics
- repetition is required for supporting different caller and callee contract semantics
  - likely to affect mainly pre and post
  - up to 2 evaluations should normally suffice,
    - other than multiple repetitions to test for unwanted CA side effects
- Implementation requirements
  - define an upper bound for repetition
- Implementation recommendations
  - allow users to configure any number of repetitions
  - make the default one single evaluation without repetitions

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- **Elision – as in skip**
  - **Only if compiler can prove that predicate:**
    - **Returns true – typically if any of the following holds:**
      - **It’s evaluated at compile time**
      - **It’s guaranteed by previous CAs (in checked terminating semantics)**
      - **Example:**
        ```
        contract_assert(x>1);
        contract_assert(x>0);      // may be elided
        ```
    - **Cannot throw, longjmp or terminate**

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- **Elision – as in rephrase**
  - **The compiler may also generate an equivalent expression**
    - **Only required to cover the well-defined cases**
      - **UB in original predicate is fair game**
  - **Existing side effects may be ditched in the process – all or nothing**
  - **No new side effects may be introduced**

Principle 6: No Destructive Side Effects
Contract assertions whose predicates, when evaluated, could affect the correctness of the program should not be supported.

```
int i = 0;
void f() pre ((++i, true));
void g() {
  f();  // i may be 0, 1, 17, etc.
}
```

The Contract-Violation Handler – you can't avoid termination (or can you?)

- **definition**
  - **a function named ::handle_contract_violation**
  - **single argument of type const std::contracts::contract_violation&**
  - **returns void**
  - **may be noexcept (but definitely doesn't have to be)**

The Contract-Violation Handler – you can't avoid termination (or can you?)

```
class contract_violation
{  // no user-accessible constructor; cannot be copied, moved, or assigned to
public:
    const char*                comment() const noexcept;
    contracts::detection_mode detection_mode() const noexcept;  // predicate_false, evaluation_exception
    exception_ptr              evaluation_exception() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind      kind() const noexcept;      // pre, post, assert
    source_location     location() const noexcept;
    evaluation_semantic semantic() const noexcept;  // (ignore), observe, enforce, (quick_enforce)
};
```

The Contract-Violation Handler – you can’t avoid termination (or can you?)

- properties
  - implementations must provide default handlers
    - recommended to output the *contract_violation* info and to be noexcept
  - default handler cannot be directly called by user code
  - but it can indirectly
    - *void invoke_default_contract_violation_handler(const contract_violation&); };*
    - but only from custom violation handlers, since
      - users have no way of creating *contract_violation* objects

The Contract-Violation Handler – you can't avoid termination (or can you?)

- **properties**
  - **implementations may allow the handler to be replaceable**
    - **i.e. provide a function with the same name, parameter types and return type**
      - **which for replaceable functions does not result in linker errors (ambiguous symbol)**
      - **may have a different exception specification than the default handler**
        - hence the standard library provides only an implementation but not an includable declaration of the default handler
    - **same as global operators new and delete**
    - **some platforms/implementers may consider that an unacceptable security risk – that's ok**

The Contract-Violation Handler – you can't avoid termination (or can you?)

- libraries control what contracts they contain
- application controls how contracts are handled
  - with the custom handler instrumental in their toolbox
- installed at link time , because:
  - different TUs and libs may be compiled at different times and with different toolchains
  - too risky security-wise at run time

The Contract-Violation Handler – you can’t avoid termination (or can you?)

- replacing the handler doesn’t require recompiling the whole application (and libraries)
  - otherwise it would be a major impediment to contracts adoption
- *contract_violation* object requirements
  - may only be produced by the implementation (not by user code)
  - may reside anywhere other than the heap
  - can be accessed by violation handler (inc. other objects pointed to) without lifetime concerns
- any exception in the CA itself is caught and treated by the handler as an additional detection mode

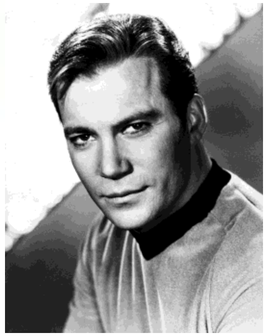
Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- Contract violation handlers are allowed to return via throw or longjmp
- Circumventing enforce semantic - a feature, not a bug!
- Bjarne himself required this in P2698R0
  - Stating that some systems cannot afford termination
- "Terminating semantics" may be a bit of a misnomer
  - Their actual guarantee (*enforce* especially) is of not allowing execution of the immediately following code
  - Most likely buggy code following a failed CA will probably crash anyway
    - Or trigger any other manifestation of UB

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- **What does noexcept(contract_assert(false)) return?**

> **Principle 1: Prime Directive**
>
> The presence or evaluation of a contract assertion in a program should not alter the correctness of that program (i.e., the property that evaluation of the program does not violate any provisions of its plain-language contract).



- Having contracts alter an exception specification would violate the prime directive
- But a previously non-throwing function/expression may now be throwing
  - Depending on whether the installed violation handler is noexcept, which is unknown at compile time
  - For which reason it must be assumed to be true (on platforms supporting throwing violation handlers)

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- Rock and hard place
  - Tell the truth and violate the prime directive
  - Or lie and be damned for all eternity
  - And we did fight over this almost for all eternity…
  - And as in other similar cases we chose to avoid the land mine altogether

> **Principle 14: Choose Ill-Formed to Enable Flexible Evolution**
>
> When no clear consensus has become apparent regarding the proper solution to a problem that Contracts could address, the relevant constructs are left ill-formed.

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- Rock and hard place
  - But rather with a little cheating
    - We made contract_assert a statement rather than an expression
    - Well, I lied a bit too - this does make noexcept(contract_assert(false)) ill-formed...
    - Now contract_assert is slightly more limited than C assert - can't be a sub-expression
    - Which immediately invoked lambdas help mitigate
      - Instead of this:
        - const int j = (contract_assert(i > 0), i);
      - We can write this:
        - const int j = ([i]{ contract_assert(i > 0); }(), i);
        - Which works since this works:
          - const int j = (void{}, i);

Final Word

- Contracts are complicated
  - As is C++
- Which is why it took so long to have them delivered
- But you won't normally care about all these details
  - Like about most other C++ details
- But you can leverage them when necessary
  - You already know what I'm going to say here...
- At the end of the day, contracts are a great tool
  - Like C++, you know the drill...

Final Word

- Many thanks to the many contributors who made Contracts finally happen in C++26!
- And special thanks to 2 of them who also thoroughly reviewed my presentation draft: Timur Doumler and Joshua Berne
  - Which makes any errors in this presentation definitely only my own

Thank You!

Let's get in touch!

andrziss@gmail.com

https://www.linkedin.com/in/andreizissu/

Questions?