Contracts for C++: Wrocław technical fixes

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)
Andrzej Krzemieński (akrzemi@gmail.com)

Document #:	P3520R0
Date:	2024-11-21
Project:	Programming Language C++
Audience:	SG21, EWG

Abstract

During CWG wording review of Contracts [P2900R11] at the November 2024 Wrocław meeting a number of minor issues were brought up. This paper discusses them and proposes resolutions.

1 Specify the mode of termination

The current wording in [P2900R11] specifies that if a contract violation occurs while evaluating a contract assertion with the enforce or quick-enforce semantic, "the program is terminated in an implementation-defined fashion".

According to CWG, this specification is insufficient to determine which modes of termination are conforming. CWG is asking to enumerate the specific modes of termination that a conforming implementation is allowed to choose between.

The following modes of termination exist in C++ today (ordered by severity):

1. `std::exit` — Normal program termination with full cleanup
2. `std::quick_exit` — Normal program termination, but with less cleanup
3. `std::_Exit` — Normal program termination with minimal cleanup
4. `std::terminate` — Abnormal program termination, the C++ way
5. `std::abort` — Abnormal program termination, the C way
6. `__builtin_trap`, `__builtin_verbose_trap`, `__fastfail`, `__debugbreak`, etc. — Implementation-defined abnormal termination modes with no cleanup

These modes of termination perform the following actions (a question mark denotes that it is implementation-defined whether the action happens):

	1	2	3	4	5	6
Can be used for normal termination that returns `0` to the host environment	✅	✅	✅	❌	❌	❌
Calls destructors of static and thread local objects	✅	❌	❌	❌	❌	❌
Calls callback functions that can be defined by the user	✅	✅	❌	✅	✅	❌
Flushes and closes streams, removes temporary files	✅	❓	❓	❓	❓	❌

The callback functions called vary by each termination mode. `std::exit` calls functions registered with `std::atexit`. `std::quick_exit` calls functions registered with `std::at_quick_exit`. `std::terminate` calls the currently installed `std::termination_handler`; the default handler calls `std::abort`. `std::abort` raises a `SIGABRT` signal which may be caught by an appropriate signal handler.

`std::exit` is called when `main` returns. `std::_Exit` is called by `std::exit` and `std::quick_exit` after they have performed their respective cleanup actions. `std::terminate` is usually called when an unrecoverable error occurs during exception handling, but has a few other notable use cases, e.g., destroying a joinable `std::thread`. `std::abort` is called by a failing `cassert`. All termination modes can be directly invoked by the user.

All termination modes except 6 call into a C or C++ standard library function, however an implementation is free to perform the actions equivalent to such a call without actually performing the call and thus without having to link in the standard library.

In all cases, termination after a contract violation results from a *defect* in the program; therefore, only the three abnormal modes of termination are appropriate. In addition, these three modes are exactly the options compiler vendors have requested or are already using for implementing of [P2900R11]. Therefore, we propose to allow exactly these options.

Proposed wording relative to [P2900R11]:

> When the program is *contract-terminated*, depending on context:
>
> — `std::terminate` is called,
>
> — `std::abort` is called,
>
> — execution is terminated.
>
> [*Note:* Performing the actions of `std::terminate` or `std::abort` without actually making a library call is a conforming implementation of contract-terminating ([intro.abstract]). — *end note*]
>
> If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is quick-enforce, the program is ~~terminated in an implementation-defined fashion~~contract-terminated. [...] If the contract-violation handler returns normally and the evaluation semantic is enforce, the program is ~~terminated in an implementation-defined fashion~~contract-terminated.

2 Disallow lambdas inside contract redeclarations

The current wording in [P2900R11] specifies that the function contract assertion sequence of a function can be repeated on a redeclaration if all repetitions consist of the same function contract specifiers in the same order. A function contract specifier $c1$ on a function declaration $d1$ is the same as a function contract specifier $c2$ on a function declaration $d2$ if they are of the same assertion kind (`pre` or `post`) and their predicates $p1$ and $p2$ would satisfy the one-definition rule (odr) if placed in an imaginary function body on the declarations $d1$ and $d2$, respectively, except the names of function parameters, names of template parameters, and the result name may be different.

This rule implies that the two lambda expressions that appear inside the predicates $p1$ and $p2$ are considered to be the same lambda expression if the odr considers the hypothetical function definitions containing those predicates to be the same:

```
void f() pre([]{ return true; }());
void f() pre([]{ return true; }());  // OK
```

However, if parameter names are involved, it might result in two lambdas that are not token-identical but would still have to be considered identical under the rule above, for example:

```
int f(int i, int j) pre([&](){ return i + j > 5 ; }());
int f(int k, int l) pre([&](){ return k + l > 5 ; }());  // OK
```

As it turns out, implementing the behaviour above requires unreasonable heroics both on the specification side and on the implementation side. If it is not sufficient to token-compare the lambdas, it becomes necessary to fully parse them, but in at least one compiler this cannot be done without creating a distinct closure type for each lambda. In addition, everywhere in the language specification lexically repeating a lambda declares a new lambda with a distinct closure type. We should not break this property.

Completely banning lambda expressions from being used in contract assertions, or completely banning the repetition of contract assertions on redeclarations seems too harsh because there are important use cases for both (see [P2890R2] and [P3066R0]). The ideal compromise is to allow the repetition of contract assertions on redeclarations, but only if they do not contain a lambda:

```
void f() pre([]{ return true; }());  // OK
void f() pre([]{ return true; }());  // error
```

Proposed wording relative to [P2900R11]:

> A *function-contract-specifier-seq* $s1$ is the same as a *function-contract-specifier-seq* $s2$ if $s1$ and $s2$ consist of the same *function-contract-specifier*s in the same order. A *function-contract-specifier* $c1$ on a function declaration $d1$ is the same as a *function-contract-specifier* $c2$ on a function declaration $d2$ if their predicates ([basic.contract.general]), $p1$ and $p2$, would satisfy the one-definition rule ([basic.def.odr]) if placed in function definitions on the declarations $d1$ and $d2$, respectively, except for renaming of parameters, renaming of template parameters, and renaming of the result name ([dcl.contract.res]), if any.
>
> [*Note:* If a *lambda-expression* is a subexpression of $p1$, and $p1$ and $p2$ are in the same translation unit, they are not the same ([expr.prim.lambda]). — *end note*]

3 Unify rules for parameters of dependent type

[P3489R0] added the rule that a parameter odr-used in a postcondition assertion needs to be explicitly declared `const`, even if it is a dependent type and the type would be `const` anyway:

```
template <typename T>
void f(T t) post (t > 0);  // error: parameter must be declared const here
```

However, we overlooked that this also excludes parameters declared via a type alias of a `const` type, which is unfortunate. For example, the following code should be well-formed:

```
using const_int_t = const int;
void f(const_int_t i) post (i > 0);  // error but should be OK
```

There are also examples where the parameter is declared via a type alias that is itself dependent. This case should arguably also be well-formed:

```
template <typename T>
void f(std::add_const_t<T> t) post(t > 0);  // error but should be OK
```

Allowing the second and third example while making the first example ill-formed would require a peculiar carve-out. In light of this information, we recommend to revert the decision to choose Option D1 from [P3489R0] and to choose Option D2 instead:

```
template <typename T>
void f(T t) post (t > 0);

int main() {
  f(1);               // error: deduced parameter type (int) is not const
  f<int>(1);          // error: parameter type (int) is not const
  f<const int>(1);    // OK
}
```

This option would also make the second and third example well-formed.

Proposed wording relative to [P2900R11]:

> If the predicate of a postcondition assertion of a function odr-uses ([basic.def.odr]) a non-reference parameter of that function, ~~all declarations of that parameter shall have a `const` qualifier~~the type of that parameter shall be `const` and shall not have array or function type.
>
> [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. The `const` qualifier of the parameter may be part of a dependent type. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*]

4 Sequences of Evaluations and Repeated Evaluations

Previously, when we adopted multiple evaluations and elision, repeated evaluations of a contract assertion could happen after any number of following contract assertions and vacuous operations — where vacuous operations included trivial initialisations and the passing of control across a function invocation boundary.

As we have been wording this requirement, the specificity of defining such vacuous operations has come under further scrutiny and it is not obvious that we can produce a very clear definition that is both correct and useful.

On the other hand, we have recently done a huge amount of work refining the mechanisms around evaluating precondition and postcondition assertions before and after parameters are passed to a function and control is transferred ([P3487R0]). Due to that work, we now have a robust specification of how function contract assertions are evaluated that freely allows that sequence of evaluations to straddle the boundary between callers and callees.

When considering the cases where we do want to allow repetition of contract assertion evaluations as an entire block, we have three primary use cases:

— When a function is invoked, all of its caller-facing preconditions are evaluated followed by all of its callee-facing preconditions.

— When a function returns, all of its callee-facing postconditions are evaluated followed by all of its caller-facing postconditions.

— Within a function's body, multiple assertion statements might be consecutive statements with no other statements between them.

In addition, there are more categories where we currently allow repetition that are not strictly needed:

— When assertion statements are at the start of a function body, we currently consider them to be part of the same sequence of assertions as the preconditions of that function.

— Assertion statements that precede a function invocation (when that function's parameters have vacuous initialisation) are considered part of the same sequence as the preconditions of that function.

— When the first action in a function body is to invoke another function, the preconditions of both the enclosing and invoked function form a contract assertion sequence.

— Assertion statements may have trivially copyable or trivially initialised variables declared between and still remain part of a single sequence.

— The postconditions of one function may be part of a sequence with the preconditions of the next invoked function, although this is limited by not being able to happen if any function parameters or temporaries are destroyed when the first function finishes its invocation.

The current proposal in [P2900R11] makes all of the above situations contract assertion sequences, and thus any contract assertion at the start of such sequences may be repeated again at the end.

We propose to instead limit contract assertion sequences to be only those cases where we clearly identify batches of contract assertions and then specify that they be evaluated at once.

Consider the following invocations:

```
void f()
  pre(pre1())
  pre(pre2())
  post(post1())
  post(post2())
{
  contract_assert(ca1());
  contract_assert(ca2());
  int i = 0;
  contract_assert(ca3(i));
}
void g()
{
  f(); // checks pre1(), pre2(), ca1(), ca2(), ca3(i), post1(), post2()
  f(); // checks pre1(), pre2(), ca1(), ca2(), ca3(i), post1(), post2()
}
```

In the above, the current rules will produce one single contract assertion sequences for the two consecutive invocations of `f`:

— `pre1()`, `pre2()`, `ca1()`, `ca2()`, `ca3(i)`, `post1()`, `post2()`, `pre1()`, `pre2()`, `ca1()`, `ca2()`, `ca3(i)`, `post1()`, `post2()`

Removing destructors for the allowed vacuous operations between contract sasertions changes this to be three contract assertion sequences due to the destruction of the automatic variable `i` that occurs when the function `f` returns:

— `pre1()`, `pre2()`, `ca1()`, `ca2()`, `ca3(i)`

— `post1()`, `post2()` `pre1()`, `pre2()`, `ca1()`, `ca2()`, `ca3(i)`

— `post1()`, `post2()`

Limitting contract assertion sequences to contract assertions that we explicitly evaluate in sequence, along with assertion statements that are consecutive further breaks this into the following 8 sequences:

— `pre1()`, `pre2()`

— `ca1()`, `ca2()`

— `ca3(i)`

— `post1()`, `post2()`

— `pre1()`, `pre2()`

— `ca1()`, `ca2()`

— `ca3(i)`

— `post1()`, `post2()`

The specification of the above sequences as sequences and allowing repetition within them is very simple and based largely on wording we have already built elsewhere for how these contract assertions are evaluated.

The particular thing we lose is the ability to take the check of `pre1()` at the start of the longer sequences and move them past, for examples, checks of `ca1()` in the first or even the second invocation of f(). This does not seem particularly compelling to retain.

Proposed wording relative to [P2900R11]:

~~*A contract assertion sequence* is a sequence of contract assertions that are consecutive.~~The following sets of contract assertions constitute a *contract assertion sequence*:

— The caller-facing precondition assertions followed by the callee-facing precondition assertions that apply to a function call;

— The callee-facing postcondition assertions followed by the caller-facing postconidtion assertions that apply to a function call;

— Assertion statements that are consecutive statements.

At any point within a contract-assertion sequence, any previously evaluated contract assertion may be evaluated again with the same or a different evaluation semantic. Such repeated evaluations of a contract assertion may happen up to an implementation-defined number of times. [Note: For example, all function contract assertions might be evaluated twice for a single function invocation, once in the caller's translation unit as part of the invoking expression and once in the callee's translation unit as part of the function definition. This allowance also extends to evaluations of contract assertions during constant evaluation. — end note]

Bibliography

[P2890R2] Timur Doumler. Contracts on lambdas. https://wg21.link/p2890R2, 2023-12-13.

[P2900R11] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r11, 2024-11-18.

[P3066R0] Timur Doumler. Allow repeating contract annotations on non-first declarations. https://wg21.link/p3066r0, 2023-12-04.

[P3487R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter that may be passed in registers. https://wg21.link/p3487r0, 2024-11-07.

[P3489R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter of dependent type. https://wg21.link/p3489r0, 2024-11-01.