Document Number: P2661R0
Date: 2022-10-14
Revises:
Reply to: Brian Bi
Bloomberg
bbi10@bloomberg.net

Miscellaneous amendments to the Contracts TS

1 Overview

We propose to make a few miscellaneous amendments to the proposed Contracts TS, [P2660R0], immediately prior to publishing it as a TS:

- Make the behavior of assumed contracts, by reference, identical to that of the `assume` attribute.
- Permit checked contracts to have arbitrary side effects, but discourage programmers from relying on the side effects by leaving unspecified the number of times they are evaluated.
- Modify the `std::experimental::contract_violation` class so that it provides access to a `std::source_location` object.
- Add a `contract_behavior` accessor to `std::experimental::contract_violation`.
- Clarify that contract conditions are not in the immediate context for SFINAE.
- Make access to non-`const`, non-reference parameters in postconditions ill-formed.
- Clarify the effect of contract conditions on defaulted and deleted functions.

2 Rationale

As explained in [P2659R0], we have deliberately kept the Contracts TS as close as possible to the wording of the C++20 working draft (prior to the removal of Contracts by [P1823R0]) after applying [P1607R1] and [P1344R1] (which were approved by EWG). This ensures that there is a starting point for the Contracts TS that is as uncontroversial as possible. However, we also believe that evolution since P1823R0 should be incorporated into the Contracts TS. In particular, we have carefully studied the recommendations made by SG21 and described in [P2521R2] and propose to adopt a number of them that we feel reflect a consensus for positive evolution.

3 Assumptions

3.1 Detailed rationale

In 2022, the `assume` attribute was added to C++23 ([P1774R8]). The Contracts TS provides a second way of expressing asssumptions, namely *via* the `assume` contract behavior. It will be surprising if these two types of assumptions do not have the same semantics. Therefore, the specification of the `assume` contract behavior should refer to that of the `assume` attribute in order to avoid any unintentional deviation.

3.2 Wording

Apply these changes to the Contracts TS [dcl.attr.contract.check]:

> The predicate of a contract with *contract-behavior* `ignore`~~ or `assume`~~ is an unevaluated operand (7.2). The predicate of a contract without *contract-behavior* where the implicit contract behavior is `ignore`~~ or `assume`~~ is not evaluated.
> [*Note:* The predicate is potentially evaluated (6.3). — *end note*]
> ~~If the predicate of a contract with the contract behavior `assume` would evaluate to `false`, the behavior is undefined.~~
>
> An assertion with contract behavior `assume` and the predicate E is equivalent to `[[assume(`E`)]]` (9.12.3). A precondition or postcondition with contract behavior `assume` and the predicate E is equivalent to a null statement (8.3) with the attribute `[[assume(`E`)]]`, where the meaning of E, the applicable semantic restrictions, and the point at which E is evaluated are as specified by 9.4.2.2. [*Note:* For example, a contract with behavior `assume` causes the odr-use (6.3) and implicit instantiation (13.9.2) of the same set of entities as the corresponding assumption. For a precondition or postcondition, name lookup for the *conditional-expression* of the corresponding assumption are done as if from the context of a function body. — *end note*]

4 Side effects in checked contracts

4.1 Detailed rationale

We believe that it desirable to permit checked contracts to have side effects without incurring undefined behavior, so that, for example, a contract condition can call a function that performs logging or allocates memory. These reasons are elaborated on by [P1670R0], which proposed to allow implementations to elide the side effects in order to prevent programmers from relying on them. SG21 also concluded that side effects should be permitted, but proposed to allow those side effects to occur more than once in order to support implementation strategies that might result in precondition and postcondition checks being placed in both the caller and callee (see P2521R2). We now propose to combine both approaches: the condition of a checked contract will be evaluated 0 or more times, and if the contract behavior is `observe` and the condition is false or would be false, then the violation handler will be invoked 1 or more times. (If the contract behavior is `enforce`, it will, of course, not be possible for the violation handler to be invoked more than once.)

4.2 Wording

Apply these changes to the Contracts TS [dcl.attr.contract.syn]:

> ~~The only side effects of a predicate of a checked contract that are allowed in a *contract-attribute-specifier* are modifications of non-volatile objects whose lifetime began and ended within the evaluation of the predicate.~~ An evaluation of a predicate that exits via an exception invokes the function `std::terminate` (14.6.2). ~~The behavior of any other side effect is undefined.~~
> ~~[*Example:* ... — *end example*]~~

Apply these changes to the Contracts TS [dcl.attr.contract.check]:

> The *violation handler* of a program is a function of type "`noexcept`$_{opt}$ function of (lvalue reference to `const std::experimental::contract_-violation`) returning `void`". The violation handler is invoked when the predicate of a checked contract evaluates to `false` or would evaluate to `false` if it were evaluated (called a *contract violation*). If the violation handler is invoked because a contract with contract behavior `observe` is violated, it is unspecified how many times the violation handler is invoked.
> [*Note:* Implementations are encouraged to ensure that the violation handler is invoked only once for each violation of a checked contract. — *end note*]
> There should be no programmatic way of setting or modifying the violation handler. [...]

[...]

A checked contract is a contract with the contract behavior `observe` or `enforce`. When the value of the predicate of a checked contract can be determined without evaluating that predicate, an implementation is allowed to omit the evaluation of the predicate, even if the predicate has side effects. If the predicate is evaluated, it is unspecified how many times it is evaluated (and, thus, how many times the side effects of the predicate occur.) [*Note:* Predicates with side effects are discouraged as the validity of a program should not depend upon the evaluation (or not) of contract predicates. — *end note*]
If the contract behavior of a violated contract is `enforce` and the execution of the violation handler does not exit via an exception, execution is terminated by invoking the function `std::terminate` (14.6.2).

5 Use of `std::source_location`

5.1 Detailed rationale

The `std::source_location` library class was adopted into C++20 in July 2019, at the same meeting at which Contracts were removed from C++20. We think that the use of `std::source_location` to represent file name, function name, and line number information corresponding to contract violations is a natural and uncontroversial step.

5.2 Wording

Replace the content of section [support.contract.cviol] of the Contracts TS with:

```
namespace std {
namespace experimental {
  class contract_violation {
  public:
    source_location source_location() const noexcept;
    string_view comment() const noexcept;
  };
}
}
```

The class `contract_violation` describes information about a contract violation generated by the implementation.

`source_location source_location() const noexcept;`
Returns: A `source_location` object representing the location where the contract violation happened (9.4.2) as if created by a call to `source_-location::current()` (17.9.2.2).

[*Note*: For the purposes of 17.9.2.2, in the case of a postcondition violation, the current function is considered to be the function specified by 9.4.2, notwithstanding the fact that `__func__` (9.5.1) may be bound to the name of a different function at the point where the postcondition lexically appears. — *end note*]

`string_view comment() const noexcept;`
Returns: Implementation-defined text describing the predicate of the violated contract.

6 `contract_violation::contract_behavior()`

6.1 Detailed rationale

We believe that the contract violation handler should have access to the contract behavior of the violated contract in order to support exponential backoff (or other backoff strategies) in logging violations of contracts with the `observe` contract behavior. If the violated contract has behavior `enforce`, the violation handler will typically want to log it unconditionally, so it needs a way to distinguish between the two. We are therefore proposing to add a `contract_behavior()` accessor to replace the `assertion_level()` accessor from P0542R5.

Note that this means the violation handler will be able to distinguish whether the implicit behavior for the entire translation unit is `observe` or `enforce`, which might seem to violate the principle that there should be no programmatic way of querying the implicit contract behavior. However, the program can only force the execution of such a query if it has control over the violation handler, so as long as there is no programmatic way of setting the violation handler, there is also no programmatic way of using the proposed `contract_behavior()` accessor to query the implicit contract behavior.

6.2 Wording

The wording below is relative to the wording for the `source_location` changes (5).

Apply these changes to the Contracts TS, [support.contract.cviol]:

```
namespace std {
namespace experimental {
  class contract_violation {
  public:
    source_location source_location() const noexcept;
    string_view comment() const noexcept;
    contract_behavior() const noexcept;
  };
}
}
```

[...]

`string_view comment() const noexcept;`
Returns: Implementation-defined text describing the predicate of the violated contract.

`string_view contract_behavior() const noexcept;`
Returns: The contract behavior of the violated contract, in the form of the string `observe` or `enforce`.

7 SFINAE and instantiation of contract conditions

7.1 Detailed rationale

In P2521R2, SG21 recommended that contract conditions should not be considered in the immediate context for SFINAE and should "behave similarly to exception specification". We agree with this direction and believe that contract conditions should be considered separately instantiated entities (like *noexcept-specifier*s) more generally and not just for the purposes of SFINAE. However, this will require more detailed study. For the time being, we are only proposing to add a non-normative note that clarifies that contract conditions are not in the immediate context for SFINAE. We believe that the normative wording of N4919 [temp.deduct]/7 already excludes contract conditions from being in the immediate context, since they are not "expressions that are used in the function type".

7.2 Wording

Add section 13.10.3.1 to the Contracts TS that adds these changes at the end of [temp.deduct.general]/7:

[*Note*: Contract conditions (9.4.2.2) are not part of the function type; therefore, substitution into contract conditions is done only when they are themselves instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*]
[*Example*:

```
#include <complex>
#include <concepts>
#include <string>

template <std::regular T>
```

```
void f(T v, T u)
[[pre observe: v < u]]; // not part of std::regular

template <typename T>
constexpr bool has_f =
  std::regular<T> &&
  requires(T v, T u) { f(v, u); };

static_assert( has_f<std::string>);          // OK: has_f
returns true
static_assert(!has_f<std::complex<float>>); // ill-formed: has_-
f causes hard instantiation error
```

— *end example*]

8 Function parameters in postconditions

8.1 Detailed rationale

The proposed Contracts TS contains wording originally from P0542R5 that makes it undefined behavior for a postcondition to odr-use a parameter value that has been modified by the function body. According to P2521R2, SG21 recommended (SF/F/N/A/SA 5/6/0/1/0) to take a different approach, in which any non-reference function parameter named by a postcondition must be declared `const` in every declaration of the function, otherwise the program is ill-formed. We therefore propose to adopt this change in the Contracts TS.

8.2 Wording

Replace [dcl.attr.contract.cond]/7 in the Contracts TS with:

> If a postcondition odr-uses (6.3) a parameter of non-reference type of the function to which it appertains, that parameter's declared type shall be `const` in every declaration of the function.
> [*Example*:

```
int f(int x)
  [[post enforce r: r == x]]       // ill-formed
{
  return ++x;
}

void g(int * p)
  [[post enforce: p != nullptr]]  // ill-formed
{
  *p = 42;
}

void h(int & r, const int x)
  [[post enforce: r == x]]          // OK
{
  r = x;
}

void h(int&, int);                  // ill-formed
```

— *end example*]

9 Contract conditions on defaulted and deleted functions

9.1 Detailed rationale

The Contracts TS does not mention the effect of contract conditions on defaulted functions. It seems that a defaulted function must become nontrivial if it has a contract condition, and there is an argument in favor of banning this entirely, since explicitly defaulting a function normally completely waives the right to explicitly specify any expressions to be evaluated when the function is function is called. However, the following example shows a use case for a contract condition on an explicitly defaulted function:

```
bool fizzable();

struct M {
  M() [[ pre : fizzable() ]] {}
};

class D {
private:
  M d_m;  // just an implementation detail

public:
  D() [[ pre : fizzable() ]] = default;
};
```

We therefore propose to continue to allow contract conditions on explicitly defaulted functions. With respect to explicitly deleted functions, it seems that contract conditions would be meaningless and probably a bug, so we propose to disallow them.

9.2 Wording

Apply these changes to the Contracts TS, [dcl.attr.contract.cond]/1:

> [...] If a friend declaration is the first declaration of the function in a translation unit and has a contract condition, the declaration shall be a definition and shall be the only declaration of the function in the translation unit.A function definition whose *function-body* is of the form `= delete ;` shall not have contract conditions. [*Note 1*: This restriction does not apply to an explicitly defaulted definition that defines the function as deleted. — *end note*]

Add section 12.4.5.2 to the Contracts TS that makes these changes to [class.default.ctor]/3:

> A default constructor is *trivial* if it is not user-provided and if:
>
> - [...]
> - for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor~~.~~, and
> - it has no contract conditions (9.4.2).
>
> Otherwise, the default constructor is *non-trivial*.

Add section 11.4.5.3 to the Contracts TS that makes these changes to [class.copy.ctor]/11:

> A copy/move constructor for class `X` is trivial if it is not user-provided and if:
>
> - [...]
> - for each non-static data member of `X` that is of class type (or array thereof), the constructor selected to copy/move that member is trivial~~;~~, and
> - it has no contract conditions (9.4.2);
>
> otherwise the copy/move constructor is *non-trivial*.

Add section 11.4.6 to the Contracts TS that makes these changes to [class.copy.assign]/9:

> A copy/move assignment operator for class `X` is trivial if it is not user-provided and if:
>
> - [...]
> - for each non-static data member of `X` that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial~~;~~, and
> - it has no contract conditions (9.4.2);
>
> otherwise the copy/move assignment operator is *non-trivial*.

Add section 11.4.7 to the Contracts TS that makes these changes to [class.dtor]/8:

A destructor is trivial if it is not user-provided and if:

- [...]
- for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor~~.~~, and
- it has no contract conditions (9.4.2).

Otherwise, the destructor is *non-trivial*.

References

[P1344R1] Nathan Myers, *Pre/Post vs. Expects/Ensures*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1344r1.md

[P1607R1] Joshua Berne *et al.*, *Minimizing Contracts*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1607r1.pdf

[P1670R0] Alisdair Meredith and Joshua Berne, *Side Effects of Checked Contracts and Predicate Elision*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1670r0.pdf

[P1774R8] Timur Doumler, *Portable assumptions*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf

[P1823R0] Nicolai Josuttis *et al.*, *Remove Contracts from C++20*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1823r0.pdf

[P2521R2] Gašper Ažman *et al.*, *Contract support—Working Paper*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2521r2.html

[P2659R0] Brian Bi, *A Proposal to Publish a Technical Specification for Contracts*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2659r0.pdf

[P2660R0] Brian Bi, *Proposed Contracts TS*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2660r0.pdf