Contracts for C++: Pre-Wrocław technical clarifications

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)

Document #:	P3483R0
Date:	2024-10-31
Project:	Programming Language C++
Audience:	SG21, EWG

Abstract

After having gained implementation and deployment experience with Contracts for C++ as proposed in [P2900R10] we identified a few corner cases for which the front matter and wording in [P2900R10] would benefit from clarifications of the design intent. In this paper, we explain the affected cases and propose to add the necessary clarifications. Importantly, we do not propose any design changes to [P2900R10].

1 Proposed clarifications

1.1 Postcondition result name with deduced type is late-parsed

As specified in [P2900R10], Section 3.4.3, if a postcondition names the return value on a non-templated function with a deduced return type, that postcondition must be attached to the declaration that is also the definition (and thus there can be no earlier declaration). This is necessary because the return type must be known in order to fully parse the postcondition predicate.

The first of two clarifications requested in the Implementers Report [P3460R0] concerns one particular aspect of this property. Consider:

```
struct A {
  template <int N> bool f() const;
};

auto h()
post (v: v.f<6>()) {
  return A{};
}
```

Should the token `<` in the postcondition predicate be parsed as the smaller-than operator, or as the opening bracket of a template argument list? This decision cannot be made without knowing the return type of `h`.

[P3460R0] considers two option: inside the postcondition predicate of `h`, its return type could be either late-parsed (i.e. the predicate expression is properly parsed only after the function body) or treated as dependent. The former option would mean that the code above is well-formed. The latter option would mean that the code above is ill-formed unless we explicitly disambiguate the expression using the `template` keyword:

```
auto h()
post (v: v.template f<6>()) {
  return A{};
}
```

[P3460R0] concludes that, in the opinion of the Clang implementers, the return type should be late-parsed, and the Standard should clarify this.

We come to the same conclusion as the implementers. Making the return type a dependent type would be strange, as `h` is not a template, and requiring explicit disambiguation would be unnecessarily user-hostile. Making the return type late parsed matches the design intent of [P2900R10], is implementable, and in fact has already been implemented in both Clang and GCC.

We believe that the proposed wording in [P2900R10] already specifies this design correctly, however to improve clarity we propose to add a note to the proposed wording and to add the above example to the front matter of [P2900R10].

It should be noted that this particular design aspect of Contracts has already been discussed in detail in [P1323R2], and the proposed solution approved by EWG. [P1323R2] uses a different example to make the same point:

```
template <typename> struct X { enum { Nested }; };
template <> struct X<int>    { struct Nested {}; };

auto f()
post (r: sizeof(X<decltype(ret)>::Nested) {  // typename needed to disambiguate?
  return 42;
}
```

The paper lists four options:

1. Disallow naming the return value in a postcondition if the function has a deduced return type.
2. Allow such naming, but treat the name of the return value as having a dependent type. This means requiring `template` and `typename` disambiguators; behaviour would be as if the point of instantiation is wherever the definition of the function occurs.
3. Allow such naming as above for templated functions, and for non-templated functions, allow such naming only for definitions. Delay the parsing of the postcondition until the return type is known is a possible implementation strategy for the non-templated function case of this option.
4. Allow such naming as above, but apply the dependent-type option for non-templated forward declarations.

The paper summarises the EWG discussion on this topic, which concludes that Option 3 (late-parsed) is the correct solution, and provides wording for Option 3. This wording has already once been approved by CWG and merged into the C++ Working Draft (for C++20, before Contracts were removed). The same wording was adopted for P2900 as well. However, through subsequent iterations of the wording in P2900, the intent of those particular sections became less clear.

To avoid future requests for clarification, we propose a few wording edits below to be more clear about what the result name does when used as an expression and that its type is always the deduced return type (with `const` qualification added) of the function, even when lexically before the point where that deduction will happen.

1.2 Trivial copies of the result object are in sequence with postconditions

The second of two clarifications requested in the Implementers Report [P3460R0] concerns the case when the return value of a function does not have an RVO slot, but is passed in a register. In this case, the result object does not exist in memory at the time the postcondition assertions are evaluated, and the implementation may instead refer to a temporary object that has the same value. The implementation may make extra copies of the result object for this purpose. These copies must be trivial, so this situation can only arise if the return type is trivially copyable.

In this case, evaluating a postcondition assertion that involves the return value requires temporary materialisation of an object that holds the return value. The question is whether the same materialised temporary must be used in each postcondition assertion:

```
int f()
post(r : ++const_cast<int&>(r) == 1)
post(r : ++const_cast<int&>(r) == 2) {  // true or false?
  return 0;
}
```

According to the specification in [P2900R10], postcondition assertions are evaluated in sequence. If the return type is *not* trivially copyable, `r` must always refer to the same object — the result object of the function. If the return type is trivially copyable, the compiler is allowed to make extra copies, but it needs to do that in sequence with evaluation of the postcondition. In other words, whether or not extra trivial copies are made cannot affect the result of the evaluation of the postcondition assertion. Therefore, in the example above, when both postconditions are evaluated once they will both evaluate to `true`.

Below, we propose adding a few words to a non-normative note as well as a code example to the wording to clarify this design intent.

1.3 For a parameter odr-used in `post`, `const` can be part of dependent type

Whether or not a function parameter is declared `const`, and can therefore be odr-used in a postcondition assertion, is not always immediately visible. Consider:

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

This function template may be instantiated with a type that is `const`-qualified, or a type that is not. However, this is not known when parsing this function template, as the variable `t` does not have a visible `const` specifier on it. It is therefore not immediately obvious when and how the above example should fail to compile.

The answer is that the parameter `t` being odr-used in the postcondition assertion has a dependent type. This situation should be treated just like other situations where dependent types occur. In particular, the above function template declaration is well-formed in its own; whether or not the postcondition assertion makes the program ill-formed should be decided at the point where the template is instantiated:

```
int main() {
  int i = 1;
  f<int>(i);        // error
  f<const int>(i); // OK
}
```

We believe that the above behaviour is already correctly specified by the wording in [P2900R10] and no other behaviour makes sense. Nevertheless, below we propose adding a few words to a non-normative note as well as a code example to the wording for additional clarity.

1.4 Lambdas can appear in redeclared pre and post sequences

Usually, when the same lambda expression is repeated token-identically, it denotes a different object that has a different type:

```
auto l1 = []{};
auto l2 = []{};
// l1 and l2 have different types

template <typename T = decltype([]{})>
struct X {};

X x1;
X x2;
// x1 and x2 have different types
```

This raises the question what should happen when a lambda appears in the predicate of a precondition or postcondition, and the affected function has a redeclaration that repeats its function contract assertion sequence (as permitted by [P2900R10], Section 3.3.1). Consider:

```
// f.h
void f() pre([]{ _ = scoped_lock(obj_mtx); return obj.is_valid(); }())

// f.cpp
void f() pre([]{ _ = scoped_lock(obj_mtx); return obj.is_valid(); }()) {
  // implementation
}
```

It seems obvious that the only possible interpretation is that in this case, unlike the previous cases, the lambda expressions must be treated as the same entity. This is essentially the same situation as having a lambda expression inside the body of an inline function that appears in multiple translation units. We should apply the same rules for what is or is not an ODR-violation in this case as well.

Below, we propose adding an example to the wording to clarify this design intent.

2 Proposed wording

The proposed wording changes are relative to [P2900R10]. Note that all proposed changes are either clarifying minor wording tweaks or clarifying non-normative notes and examples; no design changes are being proposed.

Modify [expr.prim.id.unqual] paragraph 5 as follows:

> [5] Otherwise, If the *unqualified-id* is the result name ([dcl.contract.res]) in a postcondition assertion attached to a function whose (possibly deduced, see [dcl.spec.auto]) return type is `T`, then the type of the expression is `const T`.
>
> [6] Otherwise, if the *unqualified-id* appears in the predicate of a contract assertion *C* ([basic.contract]) and the entity is
>
> — ~~the result object of (possibly deduced, see [dcl.spec.auto]) type `T` of a function call and the *unqualified-id* is the result name ([dcl.contract.res]) in a postcondition assertion, or~~
>
> — a variable declared outside of *C*, or
>
> — a structured binding of type `T` whose corresponding variable is declared outside of *C*,
>
> then the type of the expression is `const T`.

Modify [dcl.contract.func] as follows:

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-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 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:* As a result of the above, all uses and definitions of a function see the equivalent *function-contract-specifier-seq* for that function across all translation units. — *end note*] [*Example:*

```
bool b1, b2;

void f() pre (b1) pre ([]{ return b2; }());
void f();                            // OK, function-contract specifiers omitted
void f() pre (b1) pre ([]{ return b2; }());  // OK, same by ODR
void f() pre (b1);                   // error: function-contract specifiers only partially repeated
void f() pre (b1) pre (b2);          // error: not same by ODR
```

— *end example*]

[...]

If the predicate of a postcondition assertion of a function odr-uses ([basic.def.odr]) a non-reference parameter of that function, that parameter shall be declared `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 can be part of the dependent type. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:*

```
int f(const int i)
post (r: r == i);
int g(int i)
post (r: r == i); // error: i is not declared const.
int f(int i) // error: i is not declared const.
{
  return i;
}

int g(int i) // error: i is not declared const.
{
  return i;
}

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

int main() {
  int i = 1;
  f<int>(i);          // error
  f<const int>(i);    // OK
}
```

— *end example*]

Modify [dcl.contract.res] as follows:

If the implementation is permitted to introduce a temporary object for the return value ([class.temporary]), the result name may instead denote that temporary object. [*Note:* It follows that, for objects that can be returned in registers, the address of the object referred to by the result name might be a temporary materialized to hold the value before it is used to initialize the actual result object. Modifications to that temporary's value are still in sequence with the evaluation of the postcondition assertions and expected to be retained for the eventual result object. — *end note*] [*Example:*

```
int f()
  post(r : ++const_cast<int&>(r) == 1)
  post(r : ++const_cast<int&>(r) == 2) // The postcondition check is guaranteed to succeed.
{
  return 0;
}
struct A {}; // trivially copyable

struct B {    // not trivially copyable
  B() {}
  B(const B&) {}
};

template <typename T>
T f(T* ptr)
  post(r: &r == ptr)
{
  return T{};
}

int main() {
  A a = f(&a); // The postcondition check may fail.
  B b = f(&b); // The postcondition check is guaranteed to succeed.
}
```

— *end example*]

Bibliography

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. https://wg21.link/p0542r5, 2018-06-08.

[P1323R2] Hubert Tong. Contract postconditions and return type deduction. https://wg21.link/p1232r2, 2019-02-20.

[P2900R10] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r10, 2024-10-12.

[P3098R0] Timur Doumler, Gašper Ažman, and Joshua Berne. Contracts for C++: Postcondition captures. https://wg21.link/p3098r0, 2024-10-14.

[P3387R0] Timur Doumler, Joshua Berne, Iain Sandoe, and Peter Bindels. Contract assertions on coroutines. https://wg21.link/p3387r0, 2024-10-09.

[P3460R0] Eric Fiselier, Nina Ranns, and Iain Sandoe. C++ Contracts Implementers Report. https://wg21.link/p3460r0, 2024-10-16.