Document Number: P0246R0
Date: 2016-02-12
Project: ISO SC22/WG21 C++ Standard, Evolution Working Group
Authors: John Lakos, Alisdair Meredith, Nathan Myers
Reply to: Nathan Myers nmyers12@bloomberg.net

Contract Support Merged Proposal

Introduction

There has been a great deal of discussion since Kona, resulting in key simplifications as proposed here. The design that follows is a careful attempt at reconciling the most urgent concerns expressed, but is still a work in progress. The design supports:

- declarative statements of pre- and post-conditions in function headers, suitable for static analysis without access to function bodies.
- runtime checking of these statements and of additional assertions in function bodies.
- control, according to selections made at build time, of whether each check is evaluated at runtime.
- control, according to a selection made at program link time, of the programmed response to violations discovered at runtime.

This design does not include features that have been suggested but that we did not know how to specify, or that may be added without impact on the design. A few such features have been called out below, with invitations for detailed proposals. Please refer to P0247 "Criteria for Contract Support" for discussion and code examples.

The syntax and names below should be taken as placeholders. A list of open questions can be found at the end.

Glossary (to aid precise discussion)

Function Contract

> A set of statements (informally, "preconditions") about argument values, well-defined program state, and past and future events for which the effects of calling a function are well-defined, along with statements of those effects (informally, "postconditions"). The statements need not be expressible in C++, and may depend on details of the entire execution history of the program. (Example: `operator delete(void* p)` requires that `p` was obtained from

`operator new(size_t)` and not deleted since it was last so obtained.)

Widen a Contract

To alter a *function contract* such that one or more preconditions is relaxed.

Narrow a Contract

To alter a *function contract* such that an additional precondition is placed on the arguments and/or environment of a call to the function.

Precondition Expression

An executable predicate, expressed in C++ and associated with a function, that may depend on the function's argument values and on program state accessible to the caller at a call site, as evaluated at the point of the call. If the expression (were it to be evaluated at that point) would evaluate to `false` then the corresponding precondition is *violated*. Example: `[[ pre: p != 0 ]]` would be violated for null `p` at the time of the call.

Postcondition Expression

An executable predicate, expressed in C++ and associated with a function, that may depend on the values of the function's arguments and return value at the point of function return, and on program state at that point accessible to the caller. If the expression (were it to be evaluated at that point) would evaluate to `false`, then the corresponding postcondition is *violated*. Example: `[[ post: return != 0 ]]` would be violated if the function returns zero.

Assert Expression

An executable predicate *expression-statement* in C++, that could be evaluated, in context, after any lexically preceding statement, and before any following statement. If the predicate would evaluate to `false` (were it to be evaluated at that point), the assertion is *violated*. Example: `[[ assert: p != 0 ]]` would be violated if `p` were zero at this point in the program.

Check

A *precondition expression*, *postcondition expression*, or *assert expression*.

Contract Annotation

A "`pre`", "`post`", or "`assert`" attribute specifying a *check*.

Checking Level

: Either "`audit`" or "`check`", in a contract annotation, or, when compiling, as a selection of which checks (if any) to evaluate at runtime. "`check`" is the default level for any check. "`audit`" is intended for checks that would violate usability guarantees of the function, particularly "big-O" performance, relative to the running time of typical uses of the function. As a translation-time selection, the level determines which checks are evaluated and acted upon at runtime.

Specifications

1. Declarations may be annotated with precondition and postcondition *contract annotations* to support *function-contract* verification.

   Proposed syntax, by example:

   ```
   auto function(ArgType1 arg1, ArgType2 arg2, ArgType3 arg3)
       [[ pre: arg1 != 0]]
       [[ pre: arg1 < arg2]]
       [[ pre: global_predicate(arg3) ]]
       [[ post: return > 0 ]]
       [[ post: other_predicate(return, arg1) ]]
       -> ResultType;
   ```

   "`[[ assert: ... ]]`" checks are not permitted in a function *declaration-part*. Declaration-level contract annotations appear immediately after the closing parenthsesis of the *declaration-part*, and appertain to the entire declaration.

2. Function definition bodies may be similarly annotated:

   ```
   auto binary_search(RAIterator b, RAIterator e, Ordered v) -> bool
   {
       [[ assert audit: std::is_partitioned(b, e,
           [v](Value v2) { return v2 < v; }) ]];
       [[ assert: b <= e ]];
       while (b < e) {
           [[assert: *b <= *e || v < *b || *e < v ]]
           ...
       }
   }
   ```

3. Predicate expressions appearing in checks are assumed to have no side effects. Any side effect that would be caused by evaluation of such an expression might not occur when the program is executed, even when the predicate expression is specified to be checked at runtime.

4. Contract annotations in a function body may include, immediately after "`pre`",

"post", or "assert", and before the colon, a checking level. If omitted, the level is taken to be "check" (the default). The checking level designation helps to determine whether the check is evaluated at runtime.

5. All checks on declarations of identically the same function must match everywhere that the function is declared. No such relationship is assumed between overloads, between a virtual base and its overrides, or between a template and its explicit specializations. Indirect calls, whether via a function pointer, a virtual-function base-class interface, or a base-case template, are checked according to the declaration used at the call site, and also according to any checks on the declaration of the function actually called. No attempt is made to check consistency or redundancy of checks that are not required to be identical. The same check appearing on two different declarations may be evaluated once, or more than once, where evaluated at all.

6. Checks are evaluated, or not, at runtime according to the checking level specified at translation time. "audit" check expressions are evaluated only at "audit" checking level; "check" checks (including checks that do not specify a checking level) are evaluated at "audit" and "check" levels. Programs built with checking "off" evaluate no checks. Example:

```
$ cc --check-audit -c bsearch.cc    # check everything
$ cc -c bsearch.cc                  # no "audit" checks evaluated
$ cc --check-off -O -c bsearch.cc   # check nothing
```

7. If a single checking level is selected for all translation units in a program, then the above fully determines whether any given check is evaluated at runtime. Where the checking levels selected when translating different TUs *differ*, it is unspecified which of the selected levels determines, at a call site, whether to evaluate checks found in a function declaration obtained via an "#include" directive. (Implementers may choose to provide more detailed specifications for this case.) Checks specified in modules will be evaluated according to the level specified by that module for calls into it, if any, or by the level specified at translation time for calls from the TU to that module, whichever more aggressively enables runtime checking.

8. When translation units are combined, a translation unit may provide (in the manner of a user-specified operator new()) a definition of a handler that takes an argument describing the caller's context, as in N4259. The handler is called if a check is specified to be evaluated and found to be false. The response to a violation if no such handler is provided is "as if" to call std::abort(). The standard places no requirements on the values passed in the argument, but implementations are encouraged to provide informative values. Example:

```
$ nm --demangle handler.o
```

```
00000  t  contract_violation_handler(std::source_location const&)
$ cc -c qualify.o bsearch.o handler.o -o qualify
```

9. Predicates expressed in checks must be well-formed regardless of the checking level designated or selected. Names used in *precondition-* and *postcondition expressions* are looked up in the lexical context of the annotated function's body, but with no access to names not accessible by the caller. A call that violates a *precondition-* or *postcondition expression* at translation time (i.e. in a `constexpr` expression) is ill-formed. Check expressions on functions decalared `constexpr` must themselves be `constexpr`.

10. Throwing an exception from the handler called in response to a violation in calling a function identified as "`noexcept`" results in an immediate call to `terminate()`.

11. If a violation-handler function returns, execution resumes after the check. Implementations may have a build mode in which returning is not permitted, resulting in termination or UB. (We note that to transition a program to using a library with checking enabled, it is common to pass through a stage in which some checks fail, and the failure must be logged and execution resumed in order to identify more than one violation per run. When the Standard C++ Library gets annotated with checks, we expect most large programs will be found to trigger myriad violations.)

12. We do not here specify any effect of checks that are not evaluated. Implementers may provide a mode to treat checks as definitive expressions of program state for improved code generation or error checking. (Such a mode would be incompatible with allowing the violation handler to return.)

13. Check attributes are not part of the function type. Check attributes on a function pointer refer to the object, not its value. When calling through a function pointer, the checks specified on the function pointer apply in addition to any on the body of the function called.

14. Open issues:

    a. The default checking level for postcondition expressions has been suggested to be "`off`", rather than "`check`", reflecting postconditions' primary use in static analysis.

    b. Proposals are invited for standard-library convenience functions that a handler may call to implement common violation-handling policies. E.g., one that calls through a static function pointer, and another that does a `longjmp` via a named static `jmp_buf`.

    c. Proposals are invited for syntax to enable a postcondition check to specify

state, including argument values and program state, to be cached at function entry and subsequently usable in a postcondition expression.

d. Proposals are invited for syntax to specify a runtime action if a particular check is violated, in effect converting that check to part of the implementation.

e. It is our intention that implementations should be compatible with extant ABIs. The specification may need to be altered to fulfill this intention.