Questions on P2680 “Contracts for C++: Prioritizing Safety”

Timur Doumler (papers@timur.audio)
Andrzej Krzemieński (akrzemi1@gmail.com)
John Lakos (jlakos@bloomberg.net)
Joshua Berne (jberne4@bloomberg.net)
Brian Bi (bbi10@bloomberg.net)
Peter Brett (pbrett@cadence.com)
Oliver Rosten (oliver.rosten@googlemail.com)
Herb Sutter (hsutter@microsoft.com)

Document #: P2700R0
Date: 2022-11-29
Project: Programming Language C++
Audience: SG21

Abstract

In this paper, we have collated questions to the author of P2680 "Contracts for C++: Prioritizing Safety" from SG21 experts. The goal of the paper is to contribute towards a better understanding of what is being proposed in P2680 and what consequences the proposed direction would have for the C++ language, the C++ standard library, third-party libraries, compiler implementations, and users of C++.

0 Introduction

Contracts are one of the most powerful and potentially impactful C++ language feature proposals currently in development. In [P2695R0], SG21 adopted a roadmap to get a Contracts MVP into C++26. The next milestone in that roadmap is to reach consensus on how side effects should behave in a contract-checking predicate. In [P2680R0], the author advocates for a novel approach to this problem. According to that paper, contract annotations should be side-effect free when seen from outside of their cone of evaluation. In the discussion at the SG21 meeting in Kona, the author further argued that contract annotations should be free of undefined behavior (UB) by definition. According to our understanding of [P2680R0], the author's goal is to design a Contracts facility for C++ that is *safe* by construction.

However, the paper leaves a number of questions open about how this approach would work in practice and what implications this would have on users of C++. To facilitate further

discussion, we have collected questions to the author of [P2680R0] from SG21 experts. We hope that these questions will be helpful. We believe that if the author could provide answers to these questions in an updated revision of their paper, it could provide important clarification of what is being proposed, which in turn will help SG21 reach consensus on a design for Contracts.

Note that this paper, [D2700R0], is intended to complement, not replace response papers to [P2680R0] or the ongoing discussion on the SG21 mailing list.

Below, we have grouped the questions we collected into five main topics, one topic per section:

1. Use of the standard library within contract-checking predicates
2. Ramifications for third-party libraries when used within contract-checking predicates
3. Differences in behavior between the current rules of evaluation and those proposed for *safe* evaluation in contract-checking predicates
4. Specifics regarding the compile-time detection of potential UB in contract-checking predicates
5. General design goals and their implications

1 Use of the standard library

Q1.1: Given the standard library as it exists today (with no normative changes such as annotations), which standard library functions (if any) could be called in a contract-checking predicate? For example, would the following compile?

```
#include <vector>

void test(const std::vector<int> &v)
  [[ pre : v.begin() < v.end() ]] { /* ... */ }
```

Q1.2: Which (if any) normative changes to the specification of `std::vector` in the standard would have to be made to make the example above well-formed?

Q1.3: Assuming the answer to Q1.1 is “yes, it would compile”, would this code still compile with the debug Microsoft STL, where `std::vector::begin` is known to allocate? If so, by what mechanism would this call be allowed considering that memory allocation, under the rules of the framework of P2680 that have been explained so far, is never permitted in contract-checking predicates?

Q1.4: Assuming the answer to Q1.3 is “no, it would not compile”, how could someone, as part of a contract-checking predicate, construct a `std::vector` that allocates memory or otherwise causes a side effect? Would this require a different build mode? Is such a build mode being proposed?

Q1.5: Would the following example be allowed (which includes an STL algorithm, a user-provided function object, and indirection through iterators?

```
void test(auto begin, auto end)
  [[ pre: std::all_of(begin, end, is_cute{}) ]];
```

2 Third-party libraries

Q2.1: Assuming a predicate will still compile with the debug Microsoft STL, where `std::vector::begin` is known to allocate, how could a third-party library (*not* a standard library implementation) with similar behavior be written in a way that provides clients with the same guarantee of being usable in predicates while, in some builds, allocating to increase safety (similar to the Microsoft STL)?

Q2.2: If I am a 3rd party library vendor, and I provide a function like this:

```
int lib3::mul_add(int a, int b, int c)
{
  return a * b + c;
}
```

and a user of my library decides to use `lib3::mul_add` as part of their contract-checking predicates:

```
void g(int a, int b, int c)
  [[ pre: lib3::mul_add(a, b, c) < 1000 ]] { /* ... */ }
```

What would happen if I change one line in the body of `lib3::mul_add` so that it performs a contract-illegal side effect? For example,

```
int lib3::mul_add(int a, int b, int c)
{
  log("mul_add() called");
  return a * b + c;
}
```

Would g then fail to compile? If so, what would the error look like? Is there any way I can avoid breaking my users' code in this way? Would this require an additional annotation on the function, and if yes, what would such an annotation look like and how would it work?

Q2.3: If I am a 3rd party library vendor, and I provide a function `lib3::mul_add` defined as in Q2.2, what would happen if I, without changing the body of `lib3::mul_add` at all, move it from an inline function in a header into a separately compiled `.cpp` file linked in a static library? Would g then fail to compile? If so, what would the error look like? Is there any way I can avoid breaking my users' code in this way? Would this require an additional annotation on the function, and if yes, what would such an annotation look like and how would it work?

Q2.4: If I am a 3rd party library vendor, and I provide a function `lib3::mul_add` defined as in Q2.2, what would happen if I, without changing the body of `lib3::mul_add` at all, put it into a dynamic library (Windows `.dll`) that is loaded after `main()` begins but before any attempt to call `g()`? Would `g` then fail to compile? If so, what would the error look like? Is there any way I can avoid breaking my users' code in this way? Would this require an additional annotation on the function, and if yes, what would such an annotation look like and how would it work?

Q2.5: How would contracts in new code be able to make use of state validation predicates provided by pre-existing 3rd party libraries, possibly without source code and/or implemented in a different language? For example, defining a new function that uses such a library:

```
lib3::handle createLine(lib3::point start, lib3::point end)
  [[ post hnd: lib3::isValid(hnd) && lib3::isLine(hnd) ]];
```

3 New behaviors for contract-checking predicate evaluation

Q3.1: Would signed-integer math work differently in a contract-checking predicate than in the rest of the language, to avoid UB due to overflow? If so, how?

Q3.2: If signed-integer math would work differently inside a contract-checking predicate, this suggests that the same expression could evaluate differently in a contract-checking predicate than in non-contract C++ code. One consequence of this inconsistency would be that a predicate could evaluate to `true` in a contract-checking predicate, but later evaluate to `false` inside the function body. Thus, the function might be called out of contract even though the precondition check would say it is called within contract.

Here is a concrete example assuming that contract-checking predicates would have wrap-around arithmetic for type `int`.

```
int f (int i, int j)
  [[ pre : i > 0 ]]
  [[ pre : j > 0 ]]
  [[ pre : j + i < 0 ]] // possible after "wrap around" effect
                        // upon overflow, which is now legal
{
  if (i > 0)
    if (j > 0)
      if (j + i < 0) // obviously false in "old style” code,
                     // as `int`s can be assumed never to overflow
        return 0;

  _Undefined_behavior(); // UB is hit.
}
```

The inconsistency arises when we call `f` with the following parameters:

```
f(INT_MAX, 2);
```

a) Is it correct to presume that inconsistencies such as that demonstrated by the concrete example above *can* arise in practice? If not, why not?

b) Can this kind of difference in interpretation of expressions – inside vs. outside of contract-checking predicates – be a source of a different kind of *unsafety* in itself? If not, why not? If so, is that acceptable and why?

Q3.3: What other forms of UB would need to be changed to defined behavior under the P2680 framework, and why? It would be helpful to provide a reasonably comprehensive list. What behavior is proposed for each such construct, and why?

Q3.4 Is the floating-point environment within a contract-checking predicate the same as in the rest of the program?

Q3.5 Can floating-point operations within a contract-checking predicate raise floating-point exceptions and/or set `errno`?

Q3.6 If the answer to Q3.5 is yes, would the following compile?

```
void foo(double x)
  [[ pre: x/3.0 < 1.0 ]] { /* ... */ }
```

Q3.7 If the answer to Q3.6 is yes, what would happen at runtime if the combination of floating-point environment and `x` is such that `FE_INEXACT` is raised?

4 Compile-time detection of potential UB

P2680 proposes that predicates that have no side effects outside of the cone of evaluation would be evaluated with no UB. Some of the questions below concern situations in which UB might occur, or code that does not cause UB but is similar to code that can cause UB. Since we are specifying a language, we must have specific answers to all of these questions if we are to define a special mode of evaluation for contract-checking predicates. If a general model is proposed, a detailed explanation is required of how that model would deal with all the specific examples, in order to aid in understanding that model.

In each of the following questions, it should be assumed that appropriate restrictions on contract-checking predicates under the framework of P2680 would be enforced by the compiler, *i.e.*, when a question asks whether some code would compile, it is implied that the compiler would reject code that is deemed not provably safe under the framework of P2680. If the compiler would reject a piece of code, criteria that are specific enough to be checked by a compiler are required.

Q4.1: Suppose a function is given, `bool f(int)`, that has a call tree *N* function calls deep (e.g., `f` calls `f2` calls … calls `f`*N*, and each `f`*N* can call 0..*M* other functions). Assume that it is acceptable to use `f` in a contract-checking predicate because it and its whole cone of evaluation meet the requirements (definition is available, definition doesn't do illegal side effect things, etc.). Further suppose that `f` is used in a contract-checking predicate, e.g.,

```
void g(int x) [[ pre: f(x) ]] { /* ... */ }
```

Would all compilers give the same answer, that `f` is legal to be used in a contract-checking predicate, regardless of the values of *N* and *M* (the depth and width of the cone of evaluation)? Are there any limits to how deep or wide the analysis should deterministically go?

Q4.2: Would the following code compile? If not, are there (existing or proposed) annotations that could be added to make it compile? If so, what are they?

```
bool checkNotNull(int* p)
{
  int* q = p;
  return q != nullptr;
}

void f(int* p) [[ pre : checkNotNull(p) ]] {}
```

Q4.3: Assuming the answer to Q4.2 is yes, would the following compile and link? If yes, what should it do at runtime?

```
int main()
{
  int* p = nullptr;
  f(p);
  return 0;
}
```

Q4.4: Would the following code compile? If not, are there annotations that could be added to make it compile?

```
bool checkSum(int x, int y)
{
  return x + y < 100;
}

void g(int x, int y) [[ pre : checkSum(x,y) ]];
```

Q4.5: Assuming the answer to Q4.4 is yes, would the following compile? If yes, what should it do at runtime?

```
int main()
{
  g(1 << 18, 1 << 18);
}
```

Q4.6: Suppose a function `f` has a precondition C that dereferences a pointer parameter. Would the compiler be required to produce a compile-time error if it cannot be proven that, on all control flow paths on which the dereference occurs, the pointer has a provenance that makes it valid to dereference?

Q4.7: If the answer to Q4.6 is "yes", then in addition to checking the declaration of `f` to determine whether C is safe, would the compiler also be required to check all call sites of `f` and reject some *calls* to `f` (including calls that may not occur within a contract predicate) on the grounds that they may cause pointer dereference UB during the ensuing evaluation of C?

Q4.8: If the answer to Q4.7 is "no", then it would be impossible for a contract-checking predicate to ever dereference any parameter of pointer type because, although the contract-checking predicate may check for null prior to any dereference, e.g.:

```
void foo(const int* p) [[ pre: p && *p > 0 ]];
```

It seems that it is not possible, within the current language, for the contract-checking predicate to first check that `p` is not a past-the-end pointer value before dereferencing it. Is this claim correct? If not, then how so?

Q4.9: If the answer to Q4.7 is "yes", then what are the criteria for a call to `foo` as declared above to be well-formed under the framework of P2680? For example, would the call to `foo` (as defined below) below be considered well-formed? If not, why not?

```
const int array[5] = {1, 2, 3, 4, 5};

void foo(const int* p) [[ pre: p && *p > 0] ] {}

void bar(bool cond, int idx, int alt)
  [[ pre: idx >= 0 && idx < 5 ]]
  [[ pre: foo(cond ? array + idx : &alt) ]];
```

Assuming this example is well-formed, would it become ill-formed if the check `idx < 5` were replaced by `idx <= 5`? Why or why not?

Q4.10: If the answer to Q4.7 is "yes", and if the address of `foo` were taken, then, under the framework of P2680, compile-time restrictions on what could be done with such a pointer would be required, otherwise there may eventually be a call through a function pointer where it cannot be determined whether that function pointer points to a function that has a contract-checking predicate. What are the criteria for what can be done with a pointer to `foo`? For example, would all the definitions below be well-formed? If not, why not?

```
void foo(const int* p) [[ pre: p && *p > 0 ]] {}

void bar(const int*) {}

auto baz(bool cond)
{
  return [cond] { return cond ? &foo : &bar; };
}

void qux()
{
  const int array[5] = {1, 2, 3, 4, 5};
  baz(true)()(array + 4);
}
```

Assuming this example is well-formed, would it become ill-formed if the argument `array + 4` were to be replaced by `array + 5`? If so, then why?

Q4.11: Are there any other rules, not previously discussed, that are required to support the use of raw `int*` parameters in contract-checking predicates while preventing UB? If so, what are they?

Q4.12: Note that almost all implementations put a limit on allocated stack space and make it UB to have chains of invocation in a thread of execution needing more than that limit in space as storage for function parameters and automatic variables. This particular kind of UB is commonly referred to as a stack overflow, and hence UB might occur whenever invoking any non-completely-inlined function at runtime. Would the following code compile, and what would `b` do at runtime when contracts are checked?

```
bool pred(unsigned int x)
{
  if (x == 171717) { return false; }
  if (x == 343434) { return true; }
  return pred(x - 1);
}

void b(int x) [[ pre : pred(x) ]];
```

Q4.13: Note that concurrent modification of a non-atomic variable is UB. This kind of UB can occur during any read of a variable created outside the cone of evaluation of a predicate, as the compiler is unable to determine whether that variable may potentially undergo concurrent modification in another thread. With that in mind, would the following code compile and, if so, what would functions `c` and `d` do at runtime when contracts are checked?

```
bool pred(int& x)
{
  return x > 0;
}

void c(int* x) [[ pre : pred(*x) ]];
void d(int& x) [[ pre : pred(x) ]];
```

Q4.14: Note that passing a pointer to storage past the end of its duration is *implementation-defined behavior*. Would the following code compile and, if so, what would function `e` do at runtime when contracts are checked?

```
int *p1(int x)
{
  int y = x;
  return &y;
}


int *p2(int * p)
{
  int *pp = p;
  return pp;
}

bool pred(int x)
{
  int *p = p1(x);
  p = p2(p);
  return p;
}

void e(int x) [[ pre : pred(x) ]];
```

Q4.15: Note that `i = i++ + 1` has UB for primitive types. Would the following code compile and, if so, what would function `h` do at runtime when contracts are checked?

```
bool pred(int x)
{
  x = x++ + 1;
  return x > 0;
}

void h(int x) [[ pre : pred(x) ]];
```

Q4.16: Note that accessing an object past the end of its lifetime has UB. Would the following code compile and, if so, what would function `j` do at runtime when contracts are checked?

```
bool pred(int x)
{
  std::optional<int> y = x;
  if (y.value() > 34) { y.reset(); }
  return y.value() > 17;
}

void j(int x) [[ pre : pred(x) ]];
```

Q4.17: What would be the behavior if, instead of using `std::optional`, an object is first created using placement `new` in an array of bytes on the stack, then destroyed, and then accessed after the end of its lifetime?

Q4.18: Note that creating a new object within the storage of a const complete object with static storage duration is UB. Would the following code compile and, if so, what would function `k` do at runtime when contracts are checked?

```
bool pred(int x)
{
  static const int y = 17;
  new (&y) int(x);
  return y == 17;
}

void k(int x) [[ pre : pred(k) ]];
```

Q4.19: Note that invocation of `std::unreachable` is UB. Would the following code compile and, if so, what would function `l` do at runtime when contracts are checked?

```
bool pred(int x)
{
  switch (x)
  {
    case 17: case 34: case 51: return true;
    case 42: case 84: case 132: return false:
    default: std::unreachable();
  }
}

void l(int x) [[ pre : pred(x) ]];
```

Q4.20: Note that returning from a function having `[[noreturn]]` on it is UB. Would the following code compile and, if so, what would function `m` do at runtime when contracts are checked?

```
[[noreturn]] bool inner(int x)
{
  if (x % 17 == 11) { throw "hi"; }
  return false;
}

bool pred(int x)
{
  try
  {
    return inner(x);
  }
  catch (...)
  {
    return false;
  }
}
void m(int x) [[ pre : pred(x) ]];
```

Q4.21: Note that falling off the end of a function having a non-void return value is UB. Would the following code compile and, if so, what would function `n` do at runtime when contracts are checked?

```
bool pred(int x)
{
  if (x % 34 == x % 57) { return true; }
}

void n(int x) [[ pre : pred(x) ]];
```

5 General design

Q5.1: How does having no side effects outside the cone of evaluation lead to safety?

Q5.2: How does having a side effect outside the cone of evaluation reduce safety?

Q5.3: Will all compilers be expected to accept and reject (exactly) the same set of predicates? If not, how will what is accepted be determined?

Q5.4: In the proposal, what properties would be checked on a predicate at compile time? It would be helpful to provide a reasonably exhaustive list.

a) How would the checks for each property be accomplished?

b) Do these properties lead to a predicate with no UB?

c) How do these properties increase safety?

d) How does not having these properties reduce safety?

Q5.5: Consider how the restrictions on predicates that [P2680R0] proposes evolve with future changes. What are the properties of contract-checking predicates (e.g., certain kinds of non-local side effects, some forms of potential UB) that would be disallowed initially, but might become allowed in a future proposal?

For each such property:

a) What conditions would need to be met before these properties could be allowed?

b) What would stop them from being included in the initial proposal?

Q5.6: It appears that [P2680R0] proposes that the "strict" mode (where contract-checking predicates are by construction free from UB and free from side effects outside of their cone of evaluation) is the default, and a "relaxed" mode is opt-in.

a) Is this understanding correct?

b) Should only the "strict" mode be part of the Contracts MVP, or both a "strict" and a "relaxed" mode?

c) Is it technically possible to instead have only a "relaxed" mode as part of the Contracts MVP, and add the "strict" mode later as an explicit opt-in? If not, why not?

d) What are some concrete advantages of making the "strict" mode the default?

e) What, if any, are some concrete advantages of making “relaxed” the default?

Q5.7: Assuming that it is not merely contract-checking predicates themselves that need to be checked, but also the call sites (*i.e.*, Q4.7 is answered with “yes”), it is understood that there will be certain "excluded" uses of functions that have contract-checking predicates, such that, although a human might be able to prove based on analysis of the call site that the execution of the predicate is safe in a given context, the compiler is unable to make such a determination. Such “excluded” uses will thus generate a compile-time error. Does this observation, assuming it is correct, imply that the option to specify a "relaxed" contract that is free of these restrictions is an essential part of a usable Contracts feature? If not, why not?

Q5.8: Are the proposed rules to deal with the situations listed in Section 4 sufficiently precise that a compiler could check them? If so, what might be the approximate wording? If not available now, then how long might it take to provide wording for these and all other cases in

which UB might, in the absence of such compile-time checking, occur in a contract-checking predicate? In particular, would this allow us to meet the roadmap that SG21 adopted in Kona to successfully ship Contracts in C++26?

Q5.9: Does the paper propose only the prevention of side effects, or also the enforcement of idempotence (which is another characteristic of referentially transparent functions)? For instance, would the following predicate be allowed?

```
void f() [[ pre: global_counter < 100 ]];
```

Note that the expression in the precondition of `f` is side-effect-free but, if evaluated twice in a row, could return a different answer.

References

[P2695R0] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2695r0.pdf
[P2680R0] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2680r0.pdf