P2680R1 2022-12-15 Reply-To: gdr@microsoft.com
Audience: SG21

Contracts for C++: Prioritizing Safety

Gabriel Dos Reis
Microsoft

*This document suggests a design of contract predicates that emphasizes **safety by default**, by reducing opportunities for undefined behaviors in contracts and their propagation across abstraction boundaries. An accommodation, in the form of relaxed contract predicates, is provided for scenarios where "safety by default" is not a primary concern.*

1 Protecting The Protector

A precondition contract is a facility intended for mitigation against undesirable runtime behavior of a program in case of erroneous situations, or erroneous inputs, collectively going by the colloquialism "bugs" (G. Dos Reis, J. D. Garcia and F. Logozzo, et al. 2016). One source of pernicious and insidious bugs is invocation of undefined behavior (Yaghmour 2019). Pursuant to the specification of "undefined behavior", compilers are adept at exploiting any logical derivations they can glean from assuming that the input source code is free of any undefined behavior, often producing most confounding outputs when the input source code contains a slight error or programmer assumption that is formally specified as undefined behavior by the standards (Wang, et al. 2012). **Consequently, a viable contract system for C++ (however minimal) must ensure that the evaluation of a contract predicate cannot itself be exploitable source of undefined behavior (by compiler optimizers).**

Consider the following program fragment:

```
int f(int);
int g(int a) [[pre: f(a) < a ]]
{
    int r = a - f(a);
    return 2 * r;
}
```

The current (experimental) implementation of contracts in GCC (https://godbolt.org/z/Ed83v76Y1) compiles (at optimization level -O3) the definition of 'g' into what appears to be a reasonable check of the precondition, followed by the computation in the function body in case the precondition holds. When that program fragment is augmented with a definition for '`f`',

```
int f(int a) { return a + 100; }
```

GCC changes (https://godbolt.org/z/qa76xqPad) the generated code for '`g`' to unconditionally call the contract violation handler without even generating code to compute the precondition, although it

separately generates code for the definition of '`f`' to add 100 to its argument which would, in most circumstances just wrap. The reason for the newly generated code for '`g`' is because the optimizer assumes that the computation of the precondition can never overflow, otherwise the program would invoke undefined behavior. Therefore, after "looking" into the body of '`f`', it must be the case that the precondition "`a + 100 < a`" must always be false irrespective of the value of '`a`'. Note that the function '`f`' isn't even defined inline, or the precondition written literally that way. These logical derivations by the optimizer are arrived at after several layers of program transformations piercing through abstractions.

It is tempting to conclude "So what? The compiler is generating a program termination consistently on integer arithmetic overflow. So, this is good!" Not quite, and not so fast. The termination on signed integer arithmetic overflow is very much by luck – an admissible implementation of undefined behavior. It is easy to see, by changing the precondition to "`f(a) > a`". Now, the optimizer entirely eliminates the precondition check (https://godbolt.org/z/4WYTYnrMv) even though a call with argument `INT_MAX - 90` would have failed the precondition `f(a) > a` on a machine where signed integer arithmetic wraps (as would have happened if the body of `f` was executed separated on the input argument). While the standards text formally defines signed integer arithmetic overflow as invoking undefined behavior (and not wrapping), empirical evidence shows that intentional uses of wraparound behaviors are more common than is widely believed (Dietz, et al. 2012). Finally, **keep in mind that contracts are not primarily for programs that are correct, fed with correct data. They are tools we need to help guard against formally unbounded behavior in case of errors either in programs, or in data, or both**.

2 BACKGROUND

During the October 6th, 2020, teleconference of the WG21 Study Group on Contracts, SG21 took a poll to determine whether it agrees "to progress contract checking to enforce software safety first, and enable assumptions of injected facts at a later time". The result was

SF	F	N	A	SA
7	4	2	0	1

which represents a strong consensus to prioritize safety first for contracts. The "injected facts" aspect was separately progressed into C++23 via the `[[assume]]` attribute proposal (Doumler, Portable assumptions 2022). So, that left the safety aspect of contracts for SG21 to focus on. On the March 24th, 2022, teleconference review of the paper D2570R0 (Krzemienski 2022), it became clear that SG21 was stuck on the issue of side effects in contract predicates and couldn't make progress for a couple of months. Most of the conversations, issues, arguments, counterarguments, were reruns of those that were had, leading up to the C++20 Contracts proposal (G. Dos Reis, et al. 2016). The issues include "if side effects are allowed in contract predicates, how many times should an implementation evaluate those side effects?" Answers ranged from 0 to many. Furthermore, short of severely limiting implementation strategies, it was also clear that the program points at which those side-effectful contracts are evaluated impact the semantics of the program, and therefore its correctness. As such, that design is unsuited for reliable software construction at scale.

Previous attempts (G. Dos Reis, J. D. Garcia and J. Lakos, et al. 2018) that tolerated side effects in contract predicates ended up with introduction of new instances of undefined behavior, an outcome that is incompatible with a contract system designed to prioritize safety. In the case of the paper D2570R0, I

observed that as long as the side effects are not observable from the outside of the cone of evaluation (section 9.1) of the contract predicates, the whole issues of unreliability and possible new instances of undefined behavior becomes moot. That observation takes a page from the compile-time evaluation model of constant expressions in C++11 and up. While more confusion occurred, no further progress was made on the "side effects in contract predicates" issue.

The SG21 Chair reached out to me, and we had a call on June 10th, 2022, to discuss what might be paths forward on the side effects and safety issues. The conversation was very productive, and I was encouraged to produce a paper laying out in writing my concerns for SG21 to discuss, and I agreed to produce said paper for the July 15th, 2022, "papers mailing". That paper, unfortunately, could be produced only for the October 15th, 2022, mailing as P2680R0 (G. Dos Reis, Contracts for C++: Prioritizing Safety 2022). That initial version of the paper was discussed (G. Dos Reis, Contracts for C++: Prioritizing for C++ - Presentation Slides of P2680R0 2022) at the recent WG21 meeting in Kona (November 11th, 2022) as if it was a design paper as opposed to "design direction" (what I thought I was asked). In addition to confusion, the discussion revealed a few things. I was offered an opportunity to provide an updated version of P2680 (the current revision) to give answers to the compilation of rapid-fire questions in P2700R1 (Doumler, Krzemienski, et al. 2022). This revision (P2680R1) of the paper answers those questions in the Appendix. They are good questions, so it would be interesting to see what the set of answers to those questions are for the current "MVP", and how they fare with respect to the aim of SG21 for progressing contract checking to enforce software safety.

Another goal of this revision is to provide a write up of the side-effect issues, including undefined behavior, for contracts support in a systems programming language designed to promote safety by default. Hopefully, future language designers will find use for it and avoid pitfalls discovered by early explorers. It is also an answer to the call to action for improving safety in C++ programs (Stroustrup, A call to action: Think seriously about "safety"; then do something sensible about it 2022) that asks of us to "think seriously about 'safety'; and ***then*** do something sensible about it".

3 INTRODUCTION

A programming language is a set of responses to the challenges of its time, not just the output of a ruthless process of draining sprint user story tasks from a Kanban board. What are the contemporary problems that the next version of C++ needs to address? Undoubtedly, there are many. Safety should be listed among the very top. By this, I mean ***safety by default***. While it can be argued that it is possible to build – with enough care and expertise - C++ programs that are both type and resource safe, the challenge here is to provide mechanisms that robustly support scalable sound programming techniques promoting safety. Such techniques are to be practiced by millions of C++ programmers and should not require diplomas of advanced studies in language subtleties. Ideally, it should take extraordinary steps to write a program that violates memory, type, and resource safety. "Contracts" are a tool that can help the C++ community get there. But only if we can design them well enough to prioritize safety, and to underpin robust code analysis tools (including static code analysis, runtime checks, etc.)

At its core, the notion of contracts is simple. And we must keep it that simple at the language support level, so that they are recognizable by ordinary programmers – not just something we call "contracts". A contract (G. Dos Reis, et al. 2016) is generally a pre-condition or a post-condition on a function. A pre-

condition is a **predicate** that expresses the expectations of a function on its arguments. Similarly, a post-condition is a **predicate** that expresses the guarantees of a function immediately following a successful execution. It is customary to extend these aspects to include assertions inside the definition of a function: again, a **predicate** that expresses an invariant at a given program execution point.

In practice, ***contracts are summaries*** of the expectations and guarantees of a successful function call. As such, not every single action taken (or statement written) in the body of a function implementation needs to be reflected in the expression of the contracts of a function. Conversely, the ability to express the contracts for a given function is a function of the expressivity of the language available in contract predicates.

4 CONTRACT PREDICATES LANGUAGE

The expectations, and the guarantees, of a function are usually formulated in the meta language used to describe the operational semantics of the C++ programming language. For instance, an expectation of the following function

```
int deref(int* p) { return *p; }
```

is that the pointer argument represents the address of an object of type `int`. That is a precondition for the ability to dereference the pointer 'p' and reading the value at the designated location of type `int` and returning that value. There is no way to completely express that predicate as current standard C++ code. However, trace semantics for C++ can readily express that predicate.

A corollary of the above observation is that a design of a contract system for C++ will, as a necessity, exhibit inability to express all possible behavior that any arbitrary well-formed C++ program can display. This is because the meta language used to define C++ is much larger and much more expressive than the C++ language itself. It is possible to add reified fragments of that meta language to C++, e.g. a predicate `object_address`, to express the particular precondition of the `deref` function and similar functions. However, unless the object language (C++) contains the meta language used to express its semantics, there will always be swaths of inexpressible behavior as part of the contracts of a function. Consequently, serious consideration should be given to abandoning the desire of wanting to express every single aspect or behavior of a C++ program in a contract. Contracts should be summaries, not detailed transcripts of a function behavior. Those belong in the body of a function implementation.

Adding reified fragments from the meta language to the object language carries its own set of constraints, simplifications and complications as exemplified by C++20's `std::is_constant_evaluated`. The aforementioned `object_address` predicate, if conceived of as only a compile-time predicate, introduces constraints on where it can be used and how it affects invocation of a function.

In general, serious considerations should be given to a judicious set of compile-time predicates that enhances contracts support for C++ to enable robust code analysis support. I use the term "code analysis" to include "static analysis", "runtime instrumentation", and more.

5 Design Principles

The ideal that the contract predicates design presented in this paper aims for is: the evaluation of contract predicates shall be free of undefined behavior, and they shall not modify parameters they reference. Contracts provide basic mitigation framework, they should not themselves be sources of vulnerabilities. There are several sorts of causes for undefined behavior (section 7). This design does not eliminate all of them, but it is a pretty good starting point to improve upon. The impossibility of total elimination of undefined behavior from contract predicates should not be reason not to aim for a more reliable contract system. We should aim to reduce undefined behavior from contract as much as possible.

Turning on contracts in a program should only increase the reliability of the program. If the program is correct and fed with correct inputs, then there should be no difference in its behavior. The contracts should in those situations just be tautological checks. If the program depends on any possible side effects in contracts for acceptable behavior, then such side effects properly belong in the program without contracts. Consequently, emphasis should be given to contracts without side effects and without exploitable sources of undefined behavior.

This proposal modifies the current "MVP" as follows:

- Categorize contracts into two groups: (1) non-relaxed contracts; (2) relaxed contracts
- Introduction of the notion of conveyor functions

Non-relaxed contracts use the same syntax as currently defined in the "MVP", with the additional constraints that the contract predicates are designed to be free of "side effects" when their evaluations are observed from outside their cone of evaluation (section 9.1). Furthermore, the evaluation of a non-relaxed contract predicate is guaranteed free from a class of sources of undefined behavior as specified in the section 9.2 on conveyor functions.

Relaxed contract predicates are not subject to the above constraints (section 6).

6 Relaxed Contracts

This proposal suggests the modifier `relaxed` when defining preconditions and postconditions. For example, the declaration

```
int rem(int x, int y) [[ pre relaxed: (log(y), y != 0) ]]
{
    return x % y;
}
```

declares the function `rem` with a precondition contract that presumably "logs" its second operand before asserting it is nonzero.

None of the restrictions and guarantees discussed in the rest of this document applies to relaxed contracts. Of course, relaxed and non-relaxed contracts can be mixed in a function declaration. For example, the above function could have been declared as

```
int rem(int x, int y) [[ pre relaxed: (log(y), true) ]]
```

```
                                [[ pre: y != 0 ]]
    {
        return x % y;
    }
```

As a programming practice, it is recommended to separate expressions that inherently violate the restrictions for non-relaxed contracts into their own relaxed contracts, so as to maximize the guarantees of undefined behavior freedom during the evaluation of contract predicates.

7 Undefined Behavior In Contracts

The protector needs protecting, see section 1. For a usable and viable contract system for C++, the evaluation of contract predicates should be defined as guaranteed free of any form of undefined behavior.

There are a few ways to ensure that the evaluation of contract predicates is free of undefined behavior:

i. Redefine the C++ abstract machine to eliminate undefined behavior from the language entirely.
ii. Forbid operators with possible undefined behavior from the (sub)language used to express contract predicates.
iii. Tighten the specification of the abstract machine so that contract predicate evaluation never invokes undefined behavior (even if other parts might), and appropriately restrict the contract predicate language.
iv. …

Option (i) entirely eliminates the whole notion of undefined behavior and associated headaches but appears too radical a change to the language to be viable in the timeframe needed. Option (ii) preserves the abstract machine specification as is (given the nearly half century deployment of the C and C++ abstract machines), does not poke the bear of radical changes and instabilities for the existing massive codebases. Option (iii) seeks to strike a balance between options (i) and (ii), and that is what is suggested in this revision of the proposal. ***Furthermore, this freedom from undefined behavior is guaranteed only for expressions in non-relaxed contracts. Relaxed contracts are not subject to any of the restrictions described below, nor do they provide any guarantees.***

There are various sources of undefined behavior in the "core" language (Yaghmour 2019); they may be categorized into several major buckets, not two of them are dealt with the same way. Some sources of undefined behavior are restricted syntactically; others (e.g. signed arithmetic overflow) are dealt with by requiring the behavior to be implementation-defined (best) or unspecified (least optimal) instead of undefined.

7.1 Lifetime

These sources of undefined behavior pertain to accessing an object outside its lifetime or validity of a pointer. By their very nature, they are not directly syntactic. The approach suggested in this proposal is to prohibit the use of certain syntactic constructs which might – under the wrong circumstances -- lead to undefined behavior. Those restrictions are syntactic, so clearly will prohibit cases that someone might find useful.

7.1.1 Object_address

The "built-in" operator `object_address` is intended to conservatively identify pointer values that are irrefutably addresses of objects. In particular, when the expression `object_address(p)` is false, it does not necessarily follow that the value p does not designate the address of an object; that only means given the syntactic restrictions, it could not be irrefutably determined that *p* is indeed the address of an object. The expression `object_address(x)` evaluates to true if and only if:

- *x* is the expression `this`; or
- *x* is an expression of the form `&obj` where *obj* is a parameter or variable of object type, or *obj* is a parameter or variable of (possibly rvalue) reference type referring to an object type; or
- *x* is initialized with an expression *y* for which `object_address(y)` holds

It is possible to extend this definition to cover more cases (including elements of arrays, conditional addresses, etc), but for now, the specification is being kept simple in order to convey the fundamental ideas.

7.2 Arithmetic Overflow

The minimum to change to guarantee absence of undefined behavior in non-relaxed contract predicates is to say that within the evaluation of arithmetic expressions, where a violation of a precondition of a built-in arithmetic operation would lead to undefined behavior, the behavior of the program is instead unspecified. Making the behavior unspecified, instead of undefined, removes the hazards of unbounded behavior; but that still leaves some form of non-determinism in the evaluation since a compiler is not required to make the same or consistent choice for an unspecified behavior. The suggested solution here is to require the evaluation of arithmetic expression to be implementation-defined when preconditions to the built-in operators are violated. Implementation-defined behavior could include (but not limited to) a "wraparound" arithmetic exposing the underlying 2-complement representation, or a saturated arithmetic also available in modern compilers (GCC n.d.). This sort of requirement is not new since a similar restriction was added to C++ in order to evaluate expressions such as `new T[n]`, without exposing C++ programs to pernicious vulnerabilities dues to underlying integer arithmetic overflow. See Core issue CWG 624 (CWG, ISO/IEC JTC1/SC22/WG21 2008).

The exception is integer division expression `x / y` and remainder expression `x % y` where it is a compile-time error if there is no "reaching definition" of the expression `y` that is not either a non-zero constant, or is not guarded by a non-zero test equivalent to "`y != 0`".

7.3 Data Races

Race conditions are formally defined as invoking undefined behavior in the C++ standards. This proposal, at this point, does not suggest any particular solution to that problem other than reducing the "attack surface" of such sources of undefined behavior.

7.4 Aliasing

Aliasing can hide sources of undefined behavior, although they may not themselves be sources of undefined behavior, especially for certain operations on values of built-in types. For instance, consider the program fragment:

```
int baz(int& x, int& y)
{
   x = 2 * y++ + x;   // #1
   return x - y;
}
int main()
{
   int a = 76;
   return baz(a, a);  // #2
}
```

The call on line #2 to the function `bar` with two references to the same variable `a` creates an aliasing between the parameters `x` and `y` of `baz`. Therefore, the operation on line #1 formally invokes an undefined behavior since it is both modifying and reading the same variable without intervening sequence points. This proposal suggests that the result of operation be unspecified, instead of invoking an undefined behavior.

There are more general lifetime problems that can be caused by aliasing, but they are not considered in this proposal.

7.5 Incompleteness And Resource Limits

Undefined behavior of a program may also arise from incompleteness of standards text itself. That sort of undefined behavior, including those arising from executing a well-formed program beyond the resource limits of an implementation, are out of scope of this proposal.

8 Side Effects In Contracts

It is easy to design a syntactic sugar template that can expand to unstructured, arbitrary code that we would call contracts. However, to enable robust code analysis tools at scale, and scalable practice of contracts, it is necessary to put structures in place. In particular, it is necessary to take the notion of predicate more seriously, something more than an unrestricted arbitrary code block with possibly type bool. **I suggest that we make each of a pre-condition and a post-condition, a self-contained expression** (their free variables being function parameters and constants), **and side-effect free when seen from the outside of each of their cone of evaluation**. That means for example that I should not be allowed, in a pre-condition, to phone home, chat with aunties and uncles, write a log of the conversation to disk, and in the post-condition share the log with friends. If those activities are to happen as part of the proper execution of a function, then they belong in the function body proper – as we do today. Contracts are summaries of expectations of and guarantees expressed as predicates. This is not just a matter of "well, if you don't like it, don't do it". To provide safety by default, we need to go beyond the usual "live, and let live". We add programming language features to promote certain programming styles. Here, we want to promote safe programming by default using contracts. Not supporting side effects visible outside the cone of evaluation of a contract that does not limit expressivity of the language. We can already express those side effects today, easily in the body of the functions; and we do so routinely.

9 Semantics Model

It has been suggested that, in terms of code generation, a pre-condition is a prolog to a function, and that a post-condition is an epilog. While that analogy holds at the lower implementation level, it is important – from language design perspective – that not all prologs are pre-conditions, and not all epilogs are post-conditions. At the day-to-day programming level, a pre-condition is the expression of the expectations of a function. It is fundamental that such an expectation can be evaluated (symbolically if possible) by the compiler and code analysis tools, and code generated as appropriate.

I suggest abandoning the approach that a pre-condition is just a prolog, and that a post-condition is just an epilog. Those are implementation details that do not help us design a language support in the language that empowers scalable code analysis such as those based on SAL (Dos Reis, Lahiri, et al. 2014), as deployed in-the-field for over two decades.

Does that mean no side effects in contracts? The answer is: No! What I am suggesting is to take a page from the constexpr semantics model (Dos Reis and Stroustrup, General Constant Expressions for System Programming Languages 2010) (Smith 2013). We can do as much side effects as we want inside the cone of constexpr evaluation, as long as those side effects are not visible from the outside, when evaluation is finished. We wouldn't require that you call only constexpr functions, but we would require that you call only functions whose side effects stay inside the cone of evaluation of that contract. We know this model works, and we have had successful experience with it over a decade of modern C++ programming. It is easier to start from a sound solid logical ground and expand from there (as proven by the constexpr semantics model and technology) than to try to issues patches to an unprincipled and unstructured arbitrary code evaluation model. To have the hope of pretending to bring increased safety to C++, it is imperative to operate from logically sound grounds.

9.1 Cone Of Evaluation of an Expression

The evaluation of a C++ expression is a transition system, where each operation moves the execution environment from one state to another. The cone of evaluation of an expression (or a statement) is the set of (possible) state transitions spanning from the beginning state of the evaluation to the end state of evaluation of that expression.

A contract predicate shall use only conveyor functions and operators allowed in conveyor functions, with the additional restrictions that if a parameter is used to call a function, then parameter passing must be by value (not move) or by const reference; if a function parameter is used to initialize a reference or a variable local to the predicate then that initialization shall be by value (not move) or the reference shall be const-qualified. Furthermore, no modifying operator is allowed on the parameters.

9.2 Conveyor Functions

A conveyor function is conceptually a function that, when called with an argument list, performs no side effects outside of its function body or argument list. Furthermore, such a function does not perform any operation the behavior of which might invoke undefined behavior. A conveyor function is declared with the attribute `[[conveyor]]`, and its body is subject to syntactic restrictions as defined below.

Violations of those syntactic restrictions result in compile-time errors. For example, the following are perfectly good conveyor functions

```
[[conveyor]] int add(int x, int y) { return x + y; }
[[conveyor]] int inc(int& x) { return ++x; }
```

but the following definition of `deref` violates a conveyor restriction.

```
[[conveyor]] int deref(int* p)
{
   return *p; // error: 'p' is not known to be object address
}
```

It needs to be rewritten as

```
[[converyor]] int deref(int* p) [[ pre: object_address(p) ]]
{
   return *p;    // OK
}
```

9.2.1 Syntactic Definition of a Conveyor Function

If a function is declared with the attribute `[[conveyor]]`, then every redeclaration or reachable declaration of it shall be declared with the `[[conveyor]]` attribute. A conveyor function can use only built-in operations (as restricted below), or other functions declared `[[conveyor]]`, or operations inferred (at the point of use) as conveyor functions or conveyor lambda expressions. If a conveyor function or lambda has a contract, then its contract predicate shall be non-relaxed.

Note that many of the restrictions suggested here are syntactic, and first order approximations. It is possible to refine them to handle more complex cases, possibly at the expense of more complicated specifications. If your favorite use scenario is not yet handled in this framework, don't just react with rejection. Rather think if it really should, and if so what appropriate amendments can be made while preserving the general goal.

9.2.1.1 Variables

A conveyor function or lambda shall odr-use neither a variable with namespace or class scope unless that variable has a const-qualified type, nor a variable with thread-local storage. If a conveyor function or lambda odr-uses a variable with static storage duration, that variable shall have a const-qualified type.

A conveyor function shall not use an *id-expression* that either designates a *pseudo-destructor* or a destructor.

Variables defined in a conveyor function or lambda shall be explicitly initialized.

9.2.1.2 Lambda expressions

A lambda expression is a conveyor lambda if its body is either empty or is of the form `return e;` where the expression `e` is subject to the same restriction as that of inferred conveyor function.

9.2.1.3 Postfix expressions

A conveyor function or lambda cannot contain a postfix expression that is `reinterpret_cast` expression or equivalent to such an expression. The postfix expression shall not cast away `const`. The

postfix expression shall not `static_cast` from a base class to a derived class. If the postfix expression is a conversion expression then it shall not invoke a narrowing conversion. If the postfix expression is of the form *x*`->`*n* where n is a name, the *x* must be an expression for which the predicate `object_address(`*x*`)` holds, and the static type of *x* shall not be a pointer to a union type. If the postfix expression is of the form *x*`.`*n* then the static type of *x* shall not be a union type. If the postfix expression is of the form *x*`[`*y*`]` and the indexing operator is built-in, then the expression *x* shall designate an array object and the expression *y* shall be constant and be in bound of the array object.

9.2.1.4 Unary expression

A conveyor function or lambda shall not contain a unary expression that is either an *await-expression*, a *new-expression*, or a *delete-expression*. If the unary operator `*` is used then its operand *e* shall be an expression for which the predicate `object_address(`*e*`)` holds.

9.2.1.5 Explicit type conversion

A conveyor function or lambda shall not contain a *cast-expression* that semantically contains a `reinterpret_cast` subexpression.

9.2.1.6 Pointer-to-member operators

If a conveyor function or lambda contains a *pm-expression* of the form *x*`->*`*y* then the expression *x* shall be a pointer for which the predicate `object_address(`*x*`)` holds.

9.2.1.7 Multiplicative operators

If a conveyor function or lambda contains a *multiplicative-expression* of the form *x* `/` *y* or *x* `%` *y* then there shall be a reaching definition of a test equivalent to *y* `!= 0`.

9.2.1.8 Additive operators

If a conveyor function contains an *additive-expression* of the form x + y then if one of the operand is of pointer type, then it shall designate an element lexically known to be an element of an array of a constant size and the other operand shall be an integer constant and the result shall designate either an element of the array or one past the end of the array.

If a conveyor function contains an *additive-expression* of the form x – y, if both are of pointer types then they shall designate objects lexically known to be part of an array with a constant size.

9.2.1.9 Relational operators

A conveyor function or lambda shall not contain a *relational-expression* where both operands are of pointer types.

9.2.1.10 Yielding a value

A conveyor function or lambda shall not contain a *yield-expression*.

9.2.1.11 Return statement

A non-void returning conveyor function or lambda shall contain at least one return statement. If at least one control flow path of a conveyor function or lambda contains a return statement, then all exit control paths shall contain a return statement.

9.2.2 Semantics Constraints on Conveyor Functions

Conveyor functions and conveyor lambdas are either syntactically restricted or semantically restricted so that they are not themselves sources of undefined behavior (see section 7). The semantic restrictions are obtained by either defining some expressions in the context of conveyor function as not invoking undefined behavior. In practice, this restriction means that logical derivations from assumption of absence of undefined behavior cannot be propagated to drive further program transformations. The semantic restrictions enumerated in this section follow previous census of core undefined behavior (Yaghmour 2019).

When the evaluation of an arithmetic expression in a conveyor function or lambda may overflow or underflow, it is unspecified which value is returned – but implementation shall not invoke undefined behavior.

A conveyor function or lambda shall not call `std::unreachable`. A conveyor function or lambda shall not contain a throw-expression.

9.2.3 Implicit Conveyor Functions

In some cases, it is possible to infer that a function is conveyor function (like done for lambdas). For instance, a non-deleted special member function that is automatically generated is conveyor if all corresponding special member functions from base classes and from non-static data members are conveyor. Similarly an inline function the body of which is of the form return e; is conveyor if the expression e satisfies all the constraints listed in section 9.2.1; its semantics is further restricted to those listed in section 9.2.2.

10 ACKNOWLEDGMENT

I would like to thank John Spicer, the Chair of SG21, both for having run, to this day, the study group to progress “Contracts” in a way that makes sure fundamental issues are addressed (instead of rushing to produce “something”), and for reaching out to find ways to make progress on the side effects issue. José Daniel García Sánchez, Gor Nishanov, Bjarne Stroustrup, Ville Voutilainen provided valuable feedback on earlier drafts. Finally, I thank the authors of P2700 for working with me to clarify their questions, and for providing feedback on this paper.

It is not a matter of skills
But a battle of wills

-- Chuck D.

11 APPENDIX

11.1 FAQ

11.1.1 What should happen if a compiler detects UB in a contract predicate?

11.1.2 Can we write meaningful contract predicates without UB?

11.2 P2700

The presentation of P2680R0 at the Kona meeting (G. Dos Reis, Contracts for C++: Prioritizing for C++ - Presentation Slides of P2680R0 2022) sparked a set a collection of rapid-fire questions compiled into the document P2700R0 and later revised as D2700R1. This revision provides answers to D2700R1 that was accessible to me at the time of writing. Because of the shear volume of questions, and in the interest of staying faithful to the original questions, those questions are not copied here. The reader is strongly encouraged to have both documents (D2700R1 and this revision) open at the same time. This document refers to the questions using the labels they were assigned to the questions in D2700R1.

As a matter of clarification, this proposal does not design a new contract system. It specifically focuses on properties that **contract predicates** should have for a viable contract system for C++. The approach is not as novel as claimed in P2700. In fact, the approach is inspired by the decade and half experience with constexpr functions and its operational semantic model. Furthermore, a co-author of P2700 (Herb Sutter) argued recently – including at evening session at the Kona meeting on the future of C++ -- in presenting the safety challenge for the C++ community, we cannot continue doing "business as usual".

11.2.1 Use of the standard library

11.2.1.1 Q1.1

This proposal is not aiming to optimize the least amount of changes needed to existing implementations of the standard library. One of its objective functions is to maximize safety for C++ programs. If you insist on standard library implementers not doing anything today, including either taking advantage of contract predicates, or enabling their implementations to participate in non-relaxed contract predicates, or both, then the answer is "it depends on the implementation". As Herb Sutter argued recently on the safety topic, we cannot continue doing "business as usual."

An implementation of the standard library that uses (base-pointer, offset) for its `std::vector<T>::iterator` will make the program compile since the relational comparison is essentially on the offsets. An implementation that uses a single raw pointer as underlying representation of `std::vector<T>::iterator` will run afoul of the restrictions on pointer relational comparison in conveyor functions (section 9.2.1.9).

Note that the declaration of `test()` can be succinctly written as

```
#include <vector>
void test(const std::vector<int>& v)
[[ pre: not v.empty() ]] { /* … */ }
```

which I expect to compile successfully with the restrictions on conveyor functions since `std::vector<T>::empty()` is most certainly just either integer equality comparison or pointer equality comparison.

This is an example of how bringing structures to contract predicates and placing appropriate limitations on what can be expressed or how to express conditions, actually raises the level of discourse of contracts. The resulting contract is more concise and more direct than the original, low level one.

11.2.1.2 Q1.2

To make the original example compile is more implementation-dependent than language dependent. For example, an implementation of `std::vector<T>::iterator` that uses a (base-pointer, offset) pair representation can be defined in a way that is implicitly conveyor function (section 9.2.3). In fact, some existing implementations of the standard library already use similar implementation techniques.

To provide a guarantee in the standard, an "English prose" of the form "these functions, unless explicitly stated otherwise, are conveyor functions" gives guidance to implementers as to which choices they have access to, and which implementation constraints they are subject to.

For uses of standard library in non-relaxed contract predicates, an audit and a conversation with standard library implementers is needed in order to determine which facilities are appropriate (or they are willing to upgrade to) for uses in non-relaxed contract predicates. Note that the upgrade from relaxed to non-relaxed contract predicate is non-breaking for users of those facilities. Therefore, this audit and determination can be done over time, and non-blocking for the progress of the "MVP" roadmap.

11.2.1.3 Q1.3

The Microsoft implementation of `std::vector<T>::begin()` does ***not*** allocate in debug mode. The code is available for to verify: https://github.com/microsoft/STL/blob/main/stl/inc/vector. Consequently, this question is mostly based on falsehood. It is interesting that the questions focus so much on one particular implementation, with assumptions divorced from reality. For what it is worth, it is Microsoft's position that contract predicates should be free of side effects and undefined behavior.

11.2.1.4 Q1.4

See answer to question Q1.3. If the contract predicate locally constructs a vector that needs allocation (as part of its implementation) then that predicate cannot be used in a non-relaxed contracts. Note that even if you could, there are other general issues (completely independent of this proposal) whether such a predicate can be used for noexcept functions (allocation may throw).

11.2.1.5 Q1.5

Whether that is allowed depends on the value type of the iterator and associated operators (e.g. equality). See section 9.2 on conveyor functions.

11.2.2 Third-party libraries

11.2.2.1 Q2.1

See answer to question Q1.3. Microsoft STL's `std::vector<T>::begin` is ***known to never allocate***, contrary to the assertion of the question.

If a third library wants to offer no guarantees at all, then its consumers can use them only in relaxed contract predicates. If they do aim to provide usage in non-relaxed contract predicates, then they will have to audit and decide which facilities they want to provide that guarantee for. This situation is not unlike the one with constexpr, which has the community has successfully learned over time to use to provide more initialization or constness at compile time.

11.2.2.2 Q2.2

Unless you've documented your function lib3::mul_add as being usable in a non-relaxed contract predicates, the user of your library has no expectation of using it a non-relaxed contract predicates. They can however use it in relaxed contract predicates independently of how you evolve your function implementation. If on the other hand, you've documented your function lib3::mul_add to be usable in non-relaxed contract predicates, and you proceed to change its implementation the way you suggest then you're breaking your own API. Also see section 9.2 on conveyor functions on how to set such expectations and being kept in line by the compiler.

11.2.2.3 Q2.3

See answer to question Q2.2. Nothing is required if your description of the function is that usable only in relaxed contract predicates. For use in non-relaxed contract predicates, you have to declare your function as a conveyor function. See section 9.2 on conveyor functions.

11.2.2.4 Q2.4

See answer to question Q2.1. BTW, ".dll" is not formally recognized by the C++ standards, so we are in vendor extension territory here. But, again, explicitly declaring your function as conveyor removes dependency on implicit conveyor inference (section 9.2.3).

11.2.2.5 Q2.5

The simplest answer is "you can't with a non-relaxed predicate". You use relaxed contract predicates.

11.2.3 New behaviors for contract-checking predicate evaluation

11.2.3.1 Q3.1

The higher order bit of the restrictions on arithmetic expressions (see section 9.2.2) in non-relaxed predicate contract is to remove invocation of undefined behavior in case of signed integer arithmetic overflow. Note that "undefined behavior" does not mean that the "same" undefined behavior is observed across parts of the program containing syntactically the same expression. Similarly, constraining the dynamic semantics of arithmetic expressions in non-relaxed contract predicates is a refinement of undefined behavior, therefore consistent with existing semantics. See section 7.2 on integer arithmetic. Relaxed contract predicates are not constrained beyond what existing C++ requires.

11.2.3.2 Q3.2

There is no consistency in undefined behavior in existing C++. Evaluating to true in one part, and evaluating to false on another part is a perfectly conforming answer in existing C++, not just theoretically. With non-relaxed contract predicate, there is additional guarantee that you will always get the same answer for the evaluation of that particular contract. The same is not true for relaxed contract predicate (existing C++). See also the discussion of the examples in section 1. So, the behavior of non-relaxed contract predicate is an improvement.

11.2.3.2.1 Q3.2(a)

Yes, and the root cause for that is not the non-relaxed contract predicate semantics. Rather, the cause is the invocation of undefined behavior (outside the contract) which throws out any notion of consistency one may have. What non-relaxed contract predicates bring here is the ability to search for the cause of undefined behavior outside the contract predicate.

11.2.3.2.2 Q3.2(b)

The invocation of undefined behavior outside the non-relaxed contract predicate throws out any sort of guarantee one may have. That observation, in itself, is no reason not to try to have the evaluation of contracts provide more guarantees than unrestricted behavior. If you want to contain the effect of signed integer arithmetic overflow outside contracts, see paper P2687R0 (Stroustrup and Dos Reis, Design Alternatives for Type-and-Resource Safe C++ 2022). Note that proposal does not subsume this one, nor does this proposal preclude that other proposal. They are complementary.

11.2.3.3 Q3.3

See section 7 for discussion on sources of undefined behavior and the approach taken for non-relaxed contract predicates. Note that relaxed contract predicates are not under any additional restrictions.

11.2.3.4 Q3.4

In relaxed contract predicates, “yes”. In non-relaxed contract predicate “no”. The situation is not unlike that of constexpr functions invoked in constexpr contexts.

11.2.3.5 Q3.5

In relaxed contract predicates, “yes”. In non-relaxed contract predicates, “no”. See also answer to question Q3.4.

11.2.3.6 Q3.6

Yes.

11.2.3.7 Q3.7

Implementation-defined behavior.

11.2.4 Compile-time detection of potential UB

11.2.4.1 Q4.1

The answer is “yes”, existing provision in the existing C++ standards that C++ compilers translate programs within the limits of their implementation-defined resources. The rules are compositional. This proposal does not add any new implementation defined limits. This proposal also does not and cannot compel a C++ compiler to accept a program when its resources are exceeded.

11.2.4.2 Q4.2

No. It needs **[[conveyor]]** attribute, or it needs to be defined differently if conveyor is to be inferred implicitly from use.

11.2.4.3 Q4.3

No. If CheckNotNull is changed as suggested in the answer for Q4.2, then “yes”. If modified that way, then the example of program does not violate any of the side-effect or undefined behavior issue that are

the subject of tis paper. What it does at runtime is whatever the "MVP" says it should do when a function precondition is violated.

11.2.4.4 Q4.4

Use annotation `[[conveyor]]` on checksum(). See also section 9.2.2 on arithmetic operations.

11.2.4.5 Q4.5

See section 7.2 on integer arithmetic.

11.2.4.6 Q4.6

If the contract is non-relaxed, then the precondition must include a predicate testing for the validity of the pointer using `object_address` (see section 7.1.1). If that validity cannot be determined at the point where the function 'f' is called, then it is a compile time error. See section 9.2.

11.2.4.7 Q4.7

At the call sites of 'f', when the preconditions are relaxed, the compiler is not obligated to do any additional check. When the precondition is non-relaxed, then the pointer validity check (which is a compile-time checked) must be performed at each call site of 'f'. Note that this proposal does not claim that a contract predicate is "safe" or not.

11.2.4.8 Q4.8

I don't know if the premise of this function holds (it sounds to me like it doesn't) but I am going to provide some information anyway. First, you can write the precondition in a relaxed fashion and not trigger any additional check:

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

and you can call that function with whatever you want.

Second, if you wanted to write the precondition form, you would need to write

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

and at call site, the compiler checks that the argument being used to call the function is indeed the address of an object (see section 7.1.1) based purely on the information available at the call site. If the pointer is indeed provably the address of an object then it is not a past-the-end value.

11.2.4.9 Q4.9

The declaration of `foo` is ill-formed. It should be written as suggested in answer to question Q4.8. Even, with that modification, pointer arithmetic is currently not allowed in conveyor functions. It is not possible to include range propagation but that is not included in the current suggestion. So, the declaration of bar() is also ill-formed.

11.2.4.10 Q4.10

Even independently non-relaxed predicates, e.g. in current "MVP" what the address of a function with contract should be is controversial as there are several choices and each implies some implementation strategies. There is no desire to include contracts in the type of a function, but at the same time, there is a tension regarding callbacks. Independently of non-relaxed predicates, further work is needed to determine the right choice should be.

11.2.4.11 Q4.11

As the proposal is progressing, we may likely find situations where either we need new rules and we need to change rules to accommodate certain scenarios. That is to be expected of any proposal.

11.2.4.12 Q4.12

Actually, the standards say that an implementation accepts executing a well-program only within its resource limit. Exceeding those limits is squarely outside the scope of contract predicates and this proposal.

11.2.4.13 Q4.13

The examples do not illustrate concurrent read in the contract predicates. Note that the function c() with a non-relaxed precondition needs additionally `object_address(c)`.

11.2.4.14 Q4.14

Lifetime. Implementation-defined. See section 7.1.

11.2.4.15 Q4.15

No. This violation is lexical. However, there are other situations where the violation is non-lexical, e.g. hidden through aliasing. Those situations are defined as unspecified. See section 7.4.

11.2.4.16 Q4.16

Lifetime. A conveyer function is prohibited from accessing union fields.

11.2.4.17 Q4.17

Ill-formed, since *new-expression*s are prohibited from conveyor functions.

11.2.4.18 Q4.18

Ill-formed. See answer to question Q4.17.

11.2.4.19 Q4.19

The current specification of C++23 makes execution of `std::unreachable` invoke undefined behavior. As such, it is not permitted in conveyor functions. See section 9.2.2.

11.2.4.20 Q4.20

See section 9.2 for restrictions on conveyor functions – ill-formed to throw.

11.2.4.21 Q4.21

No, pred does not satisfy the constraints on conveyor functions. See section 9.2 If you insist on still using the function that way, then you need to make the contract relaxed.

11.2.5 General design

11.2.5.1 Q5.1

This proposal is consistently mischaracterized as aimed at making contract predicate “safe”. The proposal makes no claim about “safe contract predicates”, nor is that claimed anywhere in the proposal. The proposal is to make contract predicate free of side effects and free of sources of undefined behavior.

This question needs to define what it means by “safe contract predicates”.

What this proposal is aiming that is: (1) identify necessary conditions for contract systems for C++ to bring increased safety to C++ programs. Having contracts themselves be sources of undefined behavior cannot lead to increased safety. Consequently, a contract design that aims to prioritize and increase safety must exclude undefined behavior from at least contract predicates. It is a necessary condition, not a sufficient condition. The suggestion in this paper is one way to achieve that.

11.2.5.2 Q5.2

See answer to the previous question for the fundamental issue at play.

11.2.5.3 Q5.3

For a non-relaxed contract predicates, the rules for checking acceptable predicates are the same for all compilers, and explained in section 9.2. There is no prohibition for compilers to have extensions (like for general C++ without contracts). Consequently, for a given non-relaxed contract predicate, the set of compilers accepting it is exactly the set of compilers implementing those rules.

11.2.5.4 Q5.4

The basic principle of what to exclude from non-relaxed contract predicate is as follows: if a basic operation has a runtime precondition the violation of which leads to undefined behavior, then either an unguarded application such operation is rejected at compile time (e.g. dereferencing a pointer not known to designate an object), or interpreted in such a way that it does not lead to undefined behavior (e.g. integer arithmetic). Both are refinement of what was formally source of undefined behavior in non-contract predicate contexts. Exception to this is the problem of data race, which remains a problem in its own right. Note that inability to solve the data race problem is no reason to tackle the ones that we know can be solved with the technology currently available.

11.2.5.4.1 Q5.4(a)

See sections on the restrictions on conveyor functions.

11.2.5.4.2 Q5.4(b)

Relaxed contract predicates offer no guarantees of any sort. Non-relaxed contract predicates exhibit the guarantee that they cannot be the source of undefined behavior for the properties checked. See section 9.2

11.2.5.4.3 Q5.4(c)

The non-relaxed contract predicates cannot themselves be source of undefined behavior. See section 9.2

11.2.5.4.4 Q5.4(d)

A contract system in which contract predicates themselves are sources of undefined behavior offer no improvement over the current situation in terms of safety. However, because the summaries are now available at all call sites, the practical reach (not the theoretical reach) of “time travel” code transformation (see section 1) due to undefined behavior (originating from a contract) is increased, therefore safety is reduced.

11.2.5.5 Q5.5

For each restriction in effect in this proposal (this revision) that is being considered for relaxation or removal, we need a solution that identifies an interpretation of the rules that does not introduce undefined behavior. That interpretation can be a combination of compile-time error and runtime

evaluation that does not include undefined behavior. This is essentially the model followed for the evolution of 'constexpr'.

11.2.5.5.1 Q5.5(a)

To remove a restriction on a given property, we need to have a proposed interpretation that does not include undefined behavior.

11.2.5.5.2 Q5.5(b)

Not having a solution that meets answer to the previous question.

11.2.5.6 Q5.6

This proposal (this revision) indeed suggests two sorts of contract predicates: normal contract predicates, and relaxed contract predicates. I don't call the normal contract predicates "strict" because those are the default, and the design is prioritizing safety.

11.2.5.6.1 Q5.6(a)

Essentially "yes", except that there is no "strict".

11.2.5.6.2 Q5.6(b)

The initial revision (P2680R0) suggested to start with what you call "strict" mode, but suggested expanding that set in a methodical fashion. In between the publication of the first revision and the presentation of the paper in Kona, there have been good suggestions for expansions that are incorporated in this revision (P2680R1). In particular, this revision proposes to have both modes – no just one.

11.2.5.6.3 Q5.6(c)

The relaxed contract predicates are what are in the current document that is tracking. One can argue that it is minimal in some sense. That choice does not strike me as "viable" in an environment where safety has become a top priority for the C++ community.

11.2.5.6.4 Q5.6(d)

Providing a language integrated facility for mitigating against safety issues that arise from undefined behavior, and providing easy to detect contexts for contracts that need further scrutiny.

11.2.5.6.5 Q5.6(e)

This proposal is not making a case for "relaxed" being default.

11.2.5.7 Q5.7

This proposal suggests both sorts of contracts for the MVP.

11.2.5.8 Q5.8

The rules are sufficiently precise for a compiler to check them, even if they are not CWG-proof ready. See section 9.2 More precise CWG-ready specifications will be produced after the Issaquah meeting (Feb 2022) if this direction of design is approved by the group. The design suggested in this proposal for contract predicates is compatible with the roadmap for Contracts in C++26.

11.2.5.9 Q5.9

This proposal suggests two things: (1) make non-relaxed contract predicate free of sources of undefined behavior; (2) separate relaxed predicates from non-relaxed predicates. Non-relaxed contract predicates are not guaranteed other properties than those implied by the above.

The example

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

is a valid declaration in the current proposal, if `global_counter` is a const variable or designates constant. With the assumption that global_counter is non-const, the declaration is ill-formed. See section 9.2.1.1. Otherwise, you would have to write it as

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

The declaration of g() is well-formed.

11.2.5.10 Q5.10

Yes, relaxed predicates may have side effects in their behavior. In general, those side effects result in either implementation-defined behavior, or unspecified behavior, or undefined behavior, depending on the context and form of the side effects. This proposal makes no suggestion about how many times they have to be evaluated.

12 References

CWG, ISO/IEC JTC1/SC22/WG21. 2008. *CWG 624: Overflow in calculating size of allocation.* June. https://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#624.

Dietz, Will, Peng Li, John Regehr, and Vikram Adve. 2012. "Understanding Integer Overflow in C/C++." *34th International Conference on Software Engineering (ICSE).* Zurich, Switzerland: IEEE Xplore. 760--770. doi:10.1109/ICSE.2012.6227142.

Dos Reis, G., J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. 2016. *A Contract Design.* July 11. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r1.pdf.

Dos Reis, Gabriel. 2022. "Contracts for C++: Prioritizing for C++ - Presentation Slides of P2680R0." November 11. https://isocpp.org/files/papers/P2743R0.pdf.

—. 2022. *Contracts for C++: Prioritizing Safety.* October 15. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2680r0.pdf.

Dos Reis, Gabriel, and Bjarne Stroustrup. 2010. *General Constant Expressions for System Programming Languages.* March 22. https://www.stroustrup.com/sac10-constexpr.pdf.

Dos Reis, Gabriel, J. Daniel Garcia, Francesco Logozzo, Manuel Fahndrich, and Shuvendu Lahiri. 2016. *Simple Contracts for C++.* February 15. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0287r0.pdf.

Dos Reis, Gabriel, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. 2018. "Support for contract based programming in C++." June 08. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html.

Dos Reis, Gabriel, Shuvendu Lahiri, Francesco Logozzo, Thomas Ball, and Jared Parsons. 2014. *Contracts for C++: What Are the Choices?* November 23. https://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4319.pdf.

Doumler, Timur. 2022. *Portable assumptions.* June 14. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf.

Doumler, Timur, Andrzej Krzemienski, John Lakos, Joshua Berne, Brian Bi, Peter Brett, Oliver Rosten, and herb Sutter. 2022. *Questions on P2680 "Contracts for C++: Prioritizing Safety".* December 15. https://open-std.org/JTC1/SC22/WG21/docs/papers/2022/p2700r1.pdf.

GCC, GNU Compiler Collection. n.d. *Built-in Functions to Perform Arithmetic with Overflow Checking.* https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html.

Krzemienski, Andrzej. 2022. *On side effects in contract annotations.* March. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2570r0.html.

Smith, Richard. 2013. *Relaxing constraints on constexpr functions.* April 18. https://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3652.html.

Stroustrup, Bjarne. 2022. "A call to action: Think seriously about "safety"; then do something sensible about it." *P2739R0.* ISO/IEC JTC1/SC22/WG21, December 06.

Stroustrup, Bjarne, and Gabriel Dos Reis. 2022. *Design Alternatives for Type-and-Resource Safe C++.* October 15. https://open-std.org/JTC1/SC22/WG21/docs/papers/2022/p2687r0.pdf.

Wang, Xi, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. "Undefined Behavior: What Happened to My Code?" *APSys 2012.* Seoul, S. Koera: ACM.

Yaghmour, Shafik. 2019. *Enumerating Core Undefined Behavior.* September 28. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html.