P3285R0 2024-05-15 Reply-To: gdr@microsoft.com
Audience: SG23, EWG

Contracts: Protecting The Protector

Gabriel Dos Reis
Microsoft

This paper presents a set of suggestions aimed at improving the "Contracts" facility as described in P2900R6 to bring them closer to viability, in the contemporary environments where C++ is used, especially where safety of C++ programs is a fundamental concern. The core of the suggestion is to remove or to limit the reach of unbounded behavior in contract predicate evaluation. While some of these ideas were previously presented in P2680R1, the present proposal focuses on the containment of undefined behavior for improved safety. It should be emphasized that the purpose of this proposal is not the definition of a "safe subset" or a "safe mode" of C++ -- it is not a goal of this proposal and we are not asking for it here.

1 SCOPE

This paper primarily seeks input from SG23 on design requirements for the Contracts facility (P2900) in order to improve safety in C++ programs using the proposed facility. It contains specific "poll" formulations to help with design directions.

2 INTRODUCTION

The "Contracts" facility (P2900R6) offers a mitigation mechanism against undesirable runtime behavior of a program in case of erroneous situations, or erroneous inputs collectively going by the colloquialism of "bugs". One source of pernicious and insidious bugs is invocation of undefined behavior. 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 undefined behavior, often producing confounding outputs when the input source code contains a slight error or programmer assumption that is formally categorized as invoking undefined behavior. Contract predicates are expressed as ordinary code. As such, they can themselves contain bugs or be surrounded by bugs. While we can't protect against writing bugs in general, and in particular in contract annotations, we can however provide a language specification that guards against undefined behavior in contract predicate evaluation. More specifically, **a viable contract system for C++ (however minimal) to be used at large must ensure that the evaluation of a contract predicate cannot itself be exploitable source of undefined behavior (by compiler optimizers)**. Let's also keep in mind that "Contracts" aren't as much useful for a correct program execution as they are for ones that contain bugs.

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/q1fhn9n6f)[1] 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. All that looks reasonable and expected.

When the above program fragment is augmented with the following definition for '`f`',

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

the GCC implementation, after inlining the call to '`f`' (entirely at its discretion), eliminates the precondition check (https://godbolt.org/z/jz83KbsEb). The reasoning for that check is based on the justification that signed integer arithmetic overflow is undefined behavior, and therefore assuming that no UB ever occurred, the expression `a + 100 > a` must always evaluate to true irrespective of the target machine or runtime configuration. Note that *if the check wasn't eliminated*, a call to g with argument `INT_MAX - 90` would have failed the precondition on a target machine where signed integer arithmetic is delivered as a wrapping arithmetic (as would have happened if the body of 'f' was executed separately on the input argument). That is, the mitigation measure would have been successfully activated. Most real life situations aren't as simple as the above; and, of course, most programmers don't write undefined behavior on purpose. The "Contracts" specification should not compromise the main purpose of the facility; it needs to offer that guarantee for viability.

Protecting contract evaluation from UB exploitation can take several forms. One avenue that was suggested in the past was to devise some form of "optimization barrier" in contract annotations. The approach proposed in this paper is to specify more precisely a compile-time checkable condition of a contract predicate being "free of undefined behavior" that also accommodates the conventional separate compilation model of C++. It should also be noted that the purpose of this proposal is ***not*** to define a "safe subset" of C++ -- it is not a goal of this proposal and we are not asking for it here.

Poll 1: The Contract facility (P2900) should offer some form of protection against exploitation of undefined behavior (or absence thereof) in the evaluation of contract predicates.

[1] The examples on godbolt.org use the old attribute-like notation because, at the time, of this writing the "natural notation" implementation had some stability issues.

3 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 to read 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 any 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 any 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.

4 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. This proposal 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 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. Consequently, emphasis should be given to contracts without exploitable sources of undefined behavior.

This proposal modifies the current "MVP" (P2900R6) as follows:

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

4.1 NON-RELAXED CONTRACTS

Non-relaxed contracts use the same syntax as currently defined in P2900R6, with the additional checkable constraints that the contract predicates are designed to be free of "side effects" when their evaluations are observed from outside their cone of evaluation. 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 6.2 on conveyor functions.

4.2 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(event_log(y), y != 0)
{
    return x % y;
}
```

declares the function `rem` with a precondition contract that presumably "write a log event" its second operand before checking 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(event_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.

4.3 Polls

Poll 2: Irrespective of syntax, the Contract facility (P2900) should offer provisions for non-relaxed contracts and relaxed contracts.

Poll 3: The Contract facility (P2900) should default the current notation to non-relaxed contracts.

5 Undefined Behavior in Contracts

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 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; 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.

5.1 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.

5.1.1 Object_address

The "built-in" operator `std::object_address` is intended to conservatively identify pointer values that are irrefutably addresses of objects. In particular, when the expression `std::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 `std::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 a nonstatic data member, 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 `std::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.

5.2 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`"

5.3 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.

5.4 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.

5.5 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.

6 Semantic Model

6.1 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 non-relaxed 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 the 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.

6.2 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
}
```

6.2.1 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.

6.2.1.1 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.

6.2.1.2 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.

6.2.1.3 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 `std::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.

6.2.1.4 Unary Expressions

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 `std::object_address(e)` holds.

6.2.1.5 Explicit Type Conversion

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

6.2.1.6 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 `std::object_address(x)` holds.

6.2.1.7 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`.

6.2.1.8 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.

6.2.1.9 Relational Operators

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

6.2.1.10 Yielding a Value

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

6.2.1.11 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.

6.2.2 Semantic 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 **Error! Reference source not found.**). 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.

6.3 Polls

Poll 4: The Contract facility (P2900) should guarantee absence of UB in non-relaxed contracts as presented in this section.