Doc. No.: P2756R0
Project: Programming Language - C++ (WG21)
Audience: SG21 Contracts
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2022-12-31

Proposal of Simple Contract Side Effect Semantics

Introduction

We propose and motivate a set of simple, intuitive and time-honored semantics for side effects in MVP contracts. In essence, we propose that contract expressions have the same side effect semantics and evaluation semantics as any other C++ expression.

Concretely, in response to open issue P2521R0/4.2, we propose NOT allowing “side effect removal” and NOT allowing “side effect duplication” of contract expressions that have side effects.

Background

C++ programs contain expressions. The standard specifies if and when an expression is evaluated during execution of a program.

Some expressions have side effects, some do not. Roughly speaking, we say an expression has side effects if it modifies an object or calls certain standard library functions (such as input/output functions), if it reads a volatile value, and/or if it modifies program state in some observable way.

When an expression that has side effects is evaluated, we say that its side effects are initiated.

Each contract contains an expression.

Therefore, a “set of semantics for contract side effects” must specify whether or not each kind of side effect is ill-formed in a contracts expression, and if not, whether it has undefined behavior, implementation-defined behavior or standard behavior. As side effects are initiated during expression evaluation, it must also specify if and when a contract’s expression is evaluated (and how many times).

Informal Specification

We propose the following set of semantics for side effects in MVP contracts:

1) The expression of a contract may have all kinds of side effects. None are ill-formed, undefined behavior or implementation-defined behavior. All have the standard behavior of any other C++ expression. The expression may odr-use and modify whatever it wants, may call any standard library function, may read a volatile value, and so on. It may do so inside or outside its "cone of evaluation".

2) There are two contract translation modes: **No_eval** and **Eval_and_abort**. These are selected between, in a global implementation-defined manner, at the start of a translation of a program.

2a) In **No_eval** translation mode, contract expressions are not evaluated, and so any side effects they have are not initiated.

2b) In **Eval_and_abort** translation mode, contract expressions are evaluated exactly once per function call for preconditions and postconditions, and once per statement execution for assertions. As usual, any side effects an expression has are initiated once per evaluation. For assertions, they are sequenced as per a statement. Preconditions have their side effects sequenced between those prior to the function call and the function body. Postconditions have their side effects sequenced between the function body and those subsequent the function call.

Note that per the as-if rule, the implementation is free to evaluate the expression in whatever fashion (or however many times or not at all) as it likes, provided that the observable behavior of the program remains the same. Therefore, the requirement of exactly one evaluation applies only to the observable side effects of the expression (if any). ie It is the side effects that must happen exactly once, with the specified sequencing. The same thing applies to other non-contract expressions - the intent is to propose that contract expressions are no different in this regard.

Formal Specification

As a wording delta on the wording proposed in P2388R4 we propose the following changes:

[dcl.correct.test]/4: **Remove** "An implementation is allowed to substitute the evaluation of a predicate P that returns to the caller with an alternative predicate that returns the same value as P but has no side effects."

[dcl.correct.test]/9: **Replace** "it is unspecified if PRE is performed one or two times" **with** "PRE is performed one time".

[dcl.correct.test]/9: **Replace** “it is unspecified if POST is performed one or two times” **with** “POST is performed one time”.

Example

Please consider the following program (using the placeholder syntax from P2521R0):

```
int x = 0;

void f() PRE(++x) {}

int main() {
    f();
    return x;
}
```

This proposal P2756R0 proposes (in agreement with P2388R4 and D2751R0) that:
- this program is not ill-formed
- it does not have undefined behavior
- in No_eval translation mode it shall output 0

This proposal P2756R0 proposes that in Eval_and_abort mode that this program shall output 1

This disagrees with P2388R4 and D2751R0 which propose that in Eval_and_abort mode it is unspecified whether this program outputs 0, 1 or 2. (In the case of 0, the expression `++x` is substituted during translation with `x+1` by the implementation.)

Motivation

1. **Simplicity**: These semantics are simple: Contract expressions are treated the same as other expressions. In **Eval**_and_abort mode expressions are **eval**uated. In **No_eval** mode, they are **no**t **eval**uated. The remainder of the specification of these semantics is actually technically superfluous - and is just there to explicitly clarify the difference between this simple proposal and the other more complicated ones.

2. **Intuitive**: These semantics match what users expect. Users will expect that contract expressions can have the same side effects as any other expression. Informed of the names of the two available translation modes, users will expect that expressions are evaluated in Eval_and_abort mode and not evaluated in No_eval mode. They will expect that when the expressions are evaluated any side effects they contain will occur once (not zero or two times), and that they will occur at the specified time. Any deviation from

these semantics will be found surprising and counter-intuitive - if the disparity is undiscovered it can easily lead to bugs.

3. **Proven**: The two translation modes and their semantics proposed here very closely match (or at least most closely match compared to other proposals) those already in extensive use by many existing assertion facilities like the standard C assert. The proposed semantics of No_eval very closely resembles the behavior of C assert with NDEBUG defined (although contracts do syntax check the expressions in No_eval mode, whereas NDEBUG preprocesses out the expression), in that in both cases the expression is not evaluated. The proposed semantics of Eval_and_abort very closely resembles the default behavior of C assert, which evaluates the expression, it may have any side effects, and those side effects are initiated once per invocation.

4. **Ease-of-use**: It is easier to use contracts if you can put side effects in contracts expressions and depend on them occurring exactly once in Eval_and_abort mode. If you have to account for them maybe occurring zero or two times, it makes it harder to follow the behavior of the program, because there is no longer a dependable one-to-one relationship between function calls and the side effects of their contract expressions.

FAQ

Why do you even want side effects in contracts in the first place? Aren't they bad?

One kind of contract expression is like a mathematical predicate. It is a pure function of the parameters, return value and other variables / program state. It doesn't have side effects in its final form. That kind of contract expression is perfectly valid and may even turn out to be the majority kind. (We however note, in passing, that even that kind of contract may benefit from printf-debugging or incrementing a counter or other kinds of temporary side effects during development.)

Another different kind of contract expression is like a test. It describes a set of steps (like a recipe) that must be taken to get a final result of pass or fail. In conducting the steps of the test, each may (and many times does) have formal side effects, that while they change the program state or call standard library functions or even alter the behavior of the program, the new altered program behavior still meets the programs requirements. That is, the changes the test made to the program's behavior are irrelevant.

Let's look at an example. Let's say we work for Boston Dynamics and are developing a humanoid robot. We develop a C++ library that allows high-level control of the robot and one of the functions is called `kick`:

```
class Robot {
    /*...*/

    void kick();

    /*...*/
};
```

When you call the function `kick` it causes the robot to do a fancy front karate kick with its leg.

The precondition of the `kick` function is that the robot is in a standing pose. There are many poses the robot could be in: standing, sitting down, laying down, etc. There is an enumeration `Pose` that holds these categories of poses, and in order to determine what pose the robot is in there is a function called `calculate_pose`:

```
  Pose calculate_pose();
```

that returns which pose the robot is in. What `calculate_pose` does is connect to a wall-mounted camera that is following the robot and downloads an image of the robot (reusing a cached version if available or storing the new version in the cache if not, logging the interactions with the camera to the filesystem), it then runs the image through a computer vision machine learning algorithm which is implemented with multiple third-party libraries in a pipeline, that farms the heavy lifting off to a shared GPU farm. By analyzing the image this algorithm determines what pose the robot is in and then returns it.

So the expression of the precondition is `calculate_pose() == Pose::Standing`

By any stretch of the definition of side effects, that expression has them. We think it's a perfectly reasonable use case for contracts, and representative of a large group of real-world imperative "test-like" contract use cases.

(Further note that we can easily see why it could be bad for `calculate_pose` to be called multiple times in quick succession. Apart from the blatant and obvious performance leak involved in doing so, it could be that a precondition of the `calculate_pose` function is that the last time it was called was greater than some duration ago, because it takes time to cool down the resources it uses, which is done asynchronously. The length of the kick function is known to exceed that duration, so calling it once in kicks precondition is fine.)

Won't this rule out certain implementation strategies that may want to double evaluate the contract expression?

It has been suggested that some implementation strategies might want to evaluate a contracts expression twice in some cases. Once on the caller-side and another time on the callee-side. This is because you can guarantee evaluation on the callee-side, but because of indirect function calls, it isn't always known until runtime what contracts a function call has, so you can't guarantee evaluation on the caller-side. It has been further suggested that it is possible to generate better error messages on the caller-side, in particular that the callers source location can be obtained. The conclusion reached from connecting both of these things is that we could allow implementations to sometimes evaluate contract expressions twice.

First, most of the time stacktraces and debug information are used and available, even in production, so the callers source location (and its caller, and its caller, and so on all the way up to main or thread main) is usually available anyway.

Second, we describe a way to get the callers source location callee-side when full debug information is not available (even though it is unclear why someone would enable contracts keeping source locations but simultaneously stripping other debug information). When the implementation translates a function call that has contracts, it can store the address of the function call along with its source location in a table. The table is stored within the program image. At runtime when a contract fails callee-side, it can read the return address from its stack frame, and lookup the return address in that table. (This is kind of like a miniature version of debug information / stacktraces just for contract source locations.) Thus callee-side contract checking can be performed while still providing the callers source location.

It has been suggested that this strategy would be susceptible to being rendered inoperative by stack corruption (if the return address was overwritten by bad pointer arithmetic on stack objects, say), whereas the source location stored in read-only data and its address stored as an immediate in the machine instruction, would not be.

In todays world, preconditions are checked with assertions as the first statement of the function body and postconditions are checked with assertions as the last statement of the function body. We can easily determine how often such a situation would arise by how often the stack trace of one of these assertions failing is corrupted (notice that such stacktraces are reading a strict superset of the information from the stack that is suggested to be used in this table strategy). The answer is hardly ever. (Also note that if the stack is corrupted you have bigger problems than getting the callers versus callees source location of the current function call.)

Thirdly, we would point out that under the as-if rule the implementation is free to do this double evaluation provided the expression doesn't have side effects. Implementations regularly determine if an expression has side effects (or rather try and prove that it hasn't) and if not make them available for all kinds of optimizations.

So in summary: The benefit of this double evaluation strategy is that we can get the callers source location (in addition to the callees) when ALL of (a) the contracts expression has side effects; (b) debug information has been stripped and is unavailable; and (c) the return address of the function has been corrupted on the stack. Under our proposal when all three of these things is true you can only get the callees source location (not the callers).

The claim of our proposal is that it is the better tradeoff to guarantee single side effect initiation (for all the benefits enumerated), then it is to support this narrowly valuable and narrowly applicable double-evaluation implementation strategy.

If we guarantee evaluation in Eval_and_abort mode, won’t it rule out future modes that only evaluate a subset of contracts?

Under our proposal the evaluation of side effects is guaranteed in Eval_and_abort mode, so it would be conceivable that someone could create a side effect in one contract that is depended upon by another contract sequenced after it. If they are both not evaluated (No_eval) this would work, and if they were both evaluated (Eval_and_abort) this would work, but if one were evaluated, and not the other, things might break.

So if the proposed semantics are accepted, people could write code like this. Then later if we added a translation mode that only evaluated a subset of contracts, that code would not be compatible with that new translation mode.

First, we should note that having code that is incompatible with a subset of translation modes is not the end of the world. The user would just be unable to use the new translation mode, or would have to port their code to the new translation mode. If we think about it, this is not too different from having to port your code from using an old language feature to using a new one (if you want to use the new feature).

Second, we think the risk is low that people will write code like that, as evidenced by the fact that instances of such usage in existing practice of assertion facilities are rare. Notice precisely the same guarantee is provided that consecutive C asserts are either both evaluated or neither, and we see very few instances of side effect dependencies between C asserts.

Thirdly, we don’t really know if we want to offer new translation modes. C assert has seemed to survive and thrive just fine for 40 years with just the two modes. It’s entirely possible that the user demand on contracts extensions will be in some totally different dimension than new translation modes.

In any case, we think most people will be able to figure out on their own that such usage would be ill-advised. For those that can’t (or those that disagree), they’ll just have to mitigate if they want to use a new translation mode. We think this is better than the alternative of leaving evaluation unspecified.

References

P2521R0 Contract support — Working Paper
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2521r0.html

P2388R4 Minimum Contract Support: either No_eval or Eval_and_abort
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2388r4.html

D2751R0 Evaluation of Checked Contract-Checking Annotations
(assuming http://wg21.link/P2751R0 once completed)