Document: P0681r0
Date: 2017-06-16
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Evolution

Precise Semantics for Assertions

Lisa Lippincott

Abstract

I've seen a number of discussions of assertions devolve into confusion over the precise semantics of assertions. Some of this confusion seems to stem from the competing needs of different areas of software development, and some seems to stem from a more philosophical question: "How can an incorrect program have the defined behavior needed to report its incorrectness?"

Here, I first propose functional definitions of assertion and assertion failure without reference to any particular syntax. Second, I enumerate some of the needs an assertion mechanism should serve. Finally, I develop a precise proposal for the behavior of assertions, constructed as C++ code built from primitives based directly in the parameterized nondeterministic abstract machine of 4.6 [intro.execution].[1]

One unexpectedly deep aspect of this proposal is to explicitly allow certain parameters of the abstract machine — certain implementation-defined behaviors — to vary between translation units. This variation is discussed in section 3.

In this paper, I intend to develop a semantics for assertions, paying particular attention cases where an assertion may have side effects, exit with an exception, have undefined behavior, or otherwise interfere with the observable behavior of the abstract machine. To promote completeness and precision, I am grounding the work in three foundations: a functional definition of *assertion*; a list of needs that should be balanced by an assertion mechanism; and the fundamental operations of the C++ abstract machine. Such an approach is, of course, somewhat lengthy; please bear with me.

To begin, I propose the following functional definition of *assertion*:

> An assertion is an experiment we propose, that, if enacted, distinguishes incorrect behavior in a surrounding context, while leaving correct behavior of the surrounding context unaltered.

I like this formulation because we can reverse it to see how a correct assertion must behave: a correct assertion, even if enacted, does not interrupt the control flow or alter the data of the surrounding context.[2] It follows that a correct assertion may be freely ignored in execution, or that it may be repeated any number of times. Much of the remainder of this paper involves delimiting the situations in which an implementation may eliminate or repeat an assertion.

This definition also helps us define *failure* of an assertion. By taking "distinguishes" to mean a distinction in value or execution path, we see that, if a successful boolean assertion must produce the value `true`, an assertion may fail not only by producing `false`, but also by hanging, terminating, exiting with an exception, or otherwise not completing. Not all of these failures will be diagnosable, but they are nonetheless failures.

1 A variety of needs

An assertion is a multi-purpose tool. It may be used to delineate correct behavior; express a test condition; diagnose incorrect behavior; prevent continuation of a failing program; open an avenue for optimization; mark a point of agreement between functions; or simply document a step in a programmer's reasoning. And when it's doing none of those things, it should add no weight to the program.

[1] References to the C++ standard are based on the April 2017 draft N4659 [3].

[2] I deliberately avoid the formulation "has no side effects" here, because it would be actively harmful to prevent assertions from calling functions that might log diagnostic messages or temporarily use previously-uninitialized memory.

A single assertion may be used in many of these roles during the life of a program. And different programs — or different parts of the same program — may call for different roles. With so many roles to play, it's not surprising that there are conflicting needs governing the behavior of assertions. I've managed to identify some of these needs:

Reliable diagnosis In some contexts, failures of assertions must be reliably reported. This is especially important in testing and debugging, so that absence of failure can be concluded from absence of reported failure. To meet this need, incorrect programs (programs with failing assertions) must have defined behavior.

Reliable interruption of control flow The code immediately following an assertion often has undesirable or undefined behavior if the assertion has failed. To prevent the undesirable or undefined behavior, a failing assertion must prevent execution from continuing to the next statement. Preventing undefined behavior in this way also makes diagnosis more reliable. Again, to meet this need, incorrect programs must have defined behavior.

Conservative introduction In some contexts, it is important that introduction of assertions into a program not alter the existing behavior of the program. Testing the assertion, diagnosing a failure, or altering the control flow each could be an unacceptable change, so this need is in conflict with the needs for reliable diagnosis and reliable interruption of control flow.

Speedy execution If an assertion interrupts the control flow of a program, code following the assertion can sometimes be optimized based on the knowledge that the assertion has succeeded. But when we have confidence that the assertion will succeed, we can make a further optimization by eliminating the test. To enable this optimization, a program that would fail an assertion must have undefined behavior. Therefore, the need for speedy execution conflicts with the needs for reliable diagnosis, reliable interruption of control flow, and conservative introduction.

Small executable size Some programs must be shrunk to the smallest size possible. In these programs, the size of the assertion diagnosis information may not be negligible. Also, as with the need for speedy execution, some code size optimization opportunities only appear when tests are optimized away. Therefore, the need for small executable size conflicts with the needs for reliable diagnosis, reliable interruption of control flow, and conservative introduction.

Source obscurity Some programs and libraries are distributed in a stripped binary form in order to obscure the details of their operation. This need largely aligns with the need for small executable size.

Aloofness When we reason about the behavior of our programs, we treat assertions as experiments we may perform to distinguish correct from incorrect behavior. Reasoning about these experiments is easier when the experiments stand aloof from the program, leaving correct programs unaffected by the presence or testing of the assertion. Likewise, it is undesirable for a program, correct or not, to be able to simulate or alter diagnostic reports of assertion failure. This need largely aligns with the needs for reliable diagnosis and conservative introduction, but can conflict with the needs for speedy execution or small executable size, when the effect of optimizations can be observed within the program.

Coordinated diagnosis of failure One motivation for introducing a new assertion mechanism is that `cerr` is not seen as a universally acceptable way to report failure. This leaves a need for a failure reporting mechanism that independently-developed libraries can agree upon.

Intolerance of exceptions Some programs, like it or not, globally cannot tolerate exceptions. This intolerance constrains the ways in which reliable interruption of control flow can be achieved.

Intolerance of termination Some programs must globally avoid termination, and instead attempt to recover from all faults internally. This intolerance also constrains the ways in which reliable interruption of control flow can be achieved, to the point where reliable interruption of control flow, intolerance of exceptions, and intolerance of termination are in conflict.

While the conflicts between these needs can't be resolved, we can give the user the ability to choose a balance between these needs. The final three needs are program-wide, and a single choice must govern an entire executable program (though a different choice might govern a separate executable program built from the same source). The other needs are local, and choices may be made more locally, limited to a translation unit or a user-specified category of assertions.

2 Assertion categories

The contracts proposal [2] recognizes a need to separate assertions into levels, where each level corresponds to an assessment of the risks of testing an assertion or leaving it untested. The cost of testing an assertion, the likelihood of the assertion failing, and the cost of failure going undetected may all factor into the choice of assertion level.

Here, I will take that work as granted, and incorporate the assertion level into a wider scheme of *assertion categories*. The assertion category is the smallest unit of independent control of assertion behavior. I find no technical need for assertion categories to be ordered, but partially or totally ordering the categories may simplify the user experience.

I will leave the exact definition of the assertion categories to future (and past) discussion, but here suggest that an assertion's category might further vary based on whether it is a precondition or postcondition, or whether (as in [1]) it is claimed (expected to succeed for local reasons; existentially quantified) or posited (expected to succeed for non-local reasons; universally quantified).

The important point for this paper is that assertions of different categories may be controlled independently, but that the category of an assertion is governed by the one definition rule. In particular, if an assertion appears in an inline function, all translation units must agree on the category of the assertion, even if they assign different treatments to that category.

3 Parameters of the abstract machine

Because the needs listed in section 1 conflict with each other, we expect each user of the implementation to resolve the conflicts by making choices that emphasize the user's particular needs. Formally, these choices are bound to parameters of the abstract machine (4.6 [intro.execution] ¶2). While such parameters formally result in implementation-defined behavior, it is expected that implementations will define a mechanism that exposes these parameters to user control (say, by relating them to compiler switches).

The behavior of assertions is described below in terms of five interdependent boolean parameters:

`assertion_testing_specified`,
`assertions_tested`,
`assertions_assumed`,
`assertions_reported`, and
`assertions_prevent_continuation`.

Collectively, I refer to these parameters as the *treatment* of an assertion. In the interest of aloofness, these parameters should not be made directly inspectable by a program.

The expectation that these parameters will be set at compile time presents a problem: few users control the compilation of every translation unit in their programs — some libraries, particularly those provided by the operating system, will be compiled by others. To avoid letting library authors dictate the treatment of assertions, we must allow the treatment to vary between translation units.

This variation in the parameters of the abstract machine creates a further problem: assertions in inline functions and in function interfaces may appear in more than one translation unit. I propose that in this case, the choice of governing translation unit be left unspecified, but uniform within a single function execution or static initialization. This requires a subtle, and perhaps belated, definition:

> A *translation unit dependent* parameter of the abstract machine may be defined by the implementation to have different values associated with different translation units. In general, when implementation-defined behavior depends on such a parameter, the value of that parameter is

drawn from an unspecified translation unit. However, within a single execution span of a function or nonlocal variable, all translation unit dependent parameters must be drawn from a single translation unit defining the function or variable.

An *execution span* of a function f is a single execution of f, exclusive of functions called by f, but inclusive of the evaluation of default parameters of such functions. Likewise, an *execution span* of a nonlocal variable x is a single initialization of the variable, exclusive of functions called (including the constructor), but inclusive of the evaluation of default parameters of such functions.[3]

My intention is that these definitions do not introduce new behavior, but instead simply extend the existing behavior of compiler options to options that control specific implementation-defined behaviors.

4 Handler functions

Common handler functions can address the program-wide needs listed in section 1: coordinated diagnosis of failure; intolerance of exceptions; and intolerance of termination. While [2] introduces a single handler function to implement these common aspects of assertion behavior, I find it convenient here to separate the common aspects into three handler functions.

The assertion failure report handler is responsible for delivering diagnostic information about an assertion failure to an appropriate output. A simple handler might compose a message based on the handler's parameters, and write the message to `cerr`.

The exception report handler is responsible for delivering diagnostic information about an exception to an appropriate output. A simple handler might compose a message based on the handler's parameters and the exception being handled, and write the message to `cerr`.

The prevent continuation handler is responsible for interrupting the ordinary control flow of the program. A simple handler might throw an exception or terminate the program. This handler must be a `noreturn` function.

The need for aloofness suggests that the choice of handlers be made outside the program source (say, as a parameter to a linker). Formally, this makes the handler implementation-defined, even though our expectation is that the implementation will allow the user control over the choice of handlers. Further, programs should be discouraged, if not prevented, from examining the choice of handlers, directly invoking the handlers, or taking the address of any of the handlers.[4]

5 Primitive operations

To define the behavior of assertions precisely, I need a number of primitive operations that are not currently supplied by the language. While I think each of these would be a useful addition to the language, the purpose of this paper is not to propose that they be added at this time. Nevertheless, I've tried to use wording suitable for standardization.

5.1 Library functions

These primitive operations can be implemented as library functions, though some will be better implemented though compiler intrinsics.

[3] This definition leaves the invocation of the destructor of an exception object or nonlocal variable outside any execution span. A stronger definition might include those invocations in the same span as the corresponding construction. This could affect the behavior of preconditions and postconditions for destructors.

[4] One mechanism might be to provide a library function that, like `main`, users are forbidden to call, but which can be mapped to a user-provided function using a linker option.

5.1.1 unspecified

```
bool unspecified() noexcept
```

Result: An unspecified (4.6 [intro.execution] ¶3) boolean value. Subsequent invocations of `unspecified` need not produce the same value. [*Note:* A program may invoke this function to allow nondeterminism in the abstract machine. —*end note*]

5.1.2 undefined_behavior

```
void undefined_behavior()
```

A program that invokes this function has undefined behavior (4.6 [intro.execution] ¶4). [*Note:* The International Standard places no requirements on the behavior of such a program. A program may invoke this function to release the implementation from the requirements of the International Standard. —*end note*]

5.1.3 prevent_continuation

```
[[noreturn]] void prevent_continuation()
```

Invokes the prevent_continuation handler. [*Note:* The program has undefined behavior if the prevent_continuation handler returns. —*end note*]

5.2 Statements

These statements have behavior that depends on their grammatical context, and so cannot be described as library functions.

5.2.1 report failure

```
report failure ;
```

Invokes the assertion failure report handler in a manner appropriate for an asserted expression that has evaluated to false at the current location.

5.2.2 report exception

```
report exception ;
```

Invokes the exception report handler in a manner appropriate for an asserted expression whose evaluation has exited at the current location with the currently handled exception (18.3 [except.handle] ¶8). If there is no currently handled exception, this statement has undefined behavior.

6 Behavior of assertions

To separate the testing conditions from the response to assertion failure, I am here introducing *unconditional failure statements.* While these statements might be a helpful addition to the language, I am describing them here as a stepping stone toward full assertions.

6.1 Unconditional failure

```
fail unconditionally ;
```

A correct program does not execute this statement. The behavior of this statement is implementation-defined, and governed by three parameters of the abstract machine: `assertions_assumed`, `assertions_reported`, and `assertions_prevent_continuation`. These parameters may vary with the assertion category (`ac`) and translation unit (`tu`). The effect of the statement is equivalent to:

```
if ( assertions_assumed(ac,tu) )
    undefined_behavior();
if ( assertions_reported(ac,tu) )
    report failure;
if ( assertions_prevent_continuation(ac,tu) )
    prevent_continuation();
```

Here we see that `assertions_assumed`, when true, allows optimization based on the assumption that this statement is never reached. This represents a choice of speedy execution and small executable size over reliable diagnosis, reliable interruption of control flow, or conservative introduction.

Taking `assertions_reported` to be true represents a choice of reliable diagnosis over source obscurity, small executable size, and conservative introduction. This is perhaps the least vital of the parameters, as source obscurity is of little concern in many projects, and the penalty to executable size may be negligible in larger projects. But I understand that in some projects the latter concerns cannot be ignored, and this parameter allows programmers on such projects to avoid paying a cost for what they don't need.

Finally, taking `assertions_prevent_continuation` to be true represents a choice of reliable diagnosis and reliable interruption of control flow over conservative introduction. If control is allowed to continue to a following statement that relies on the assertion, undefined behavior might result, preventing delivery of the report of assertion failure.

The values of `assertions_reported` and `assertions_prevent_continuation` are made moot when `assertions_assumed` is true. Therefore these three binary choices combine to give `fail unconditionally` five possible behaviors, including one which has no effect other than to distinguish the execution path as failing.

6.2 Failure with an exception

```
fail exceptionally ;
```

This statement is similar to `fail unconditionally`, but is used when the failure is due to an exception. The effect of the statement is equivalent to:

```
if ( assertions_assumed(ac,tu) )
    undefined_behavior();
if ( assertions_reported(ac,tu) )
    report exception;
throw;
```

As with `fail unconditionally`, when when `assertions_assumed` is true, the value of `assertions_reported` is made moot. The parameter `assertions_prevent_continuation` never affects this statement, as the exception itself prevents continuation to the next statement.[5] Continuation to the next statement is always prevented by `fail exceptionally`.

[5] An alternate formulation has `assertions_prevent_continuation` govern invocation of yet another handler, before the exception is rethrown. This adds complexity without, in my estimation, addressing any of the needs identified in section 1.

6.3 Boolean assertions

A boolean assertion has the behavior of the following code, where the word "`expression`" is replaced by the asserted expression. The relevant parameters of the abstract machine are represented as functions of the assertion category `ac` and the translation unit `tu`.

```
if ( assertion_testing_specified(ac,tu)
        ? assertions_tested(ac,tu)
        : unspecified() )
   {
    bool failed = true;
    try
       {
        if ( expression )
            failed = false;
       }
    catch ( ... )
       {
        fail exceptionally;
        // fail exceptionally always prevents continuation
       }
    if ( failed )
        fail unconditionally;
   }
```

Note that when `assertion_testing_specified` is false, the value of `assertions_tested` is made moot. Therefore there are three choices for testing: testing unspecified, specified tested, and specified untested. In the specified untested case, the parameters governing response to failure are also made moot.

Here we see that leaving testing unspecified allows the widest latitude when optimizing for speedy execution. It allows an implementation to insert a test — even a test that may throw or have side effects — in order to separate the assertion success path (presumably a hot path) from the assertion failure path (presumably a cold path). Of course, when testing is left unspecified, reliable diagnosis, reliable interruption of control flow, and conservative introduction are compromised.

When assertions are specified to be tested, reliable diagnosis and reliable interruption of control flow are chosen over speedy execution, small executable size, or conservative introduction.

Finally, when assertions are specified to be untested, conservative introduction is prioritized over all other concerns, leaving the assertion with no effect on the defined and specified behavior of the program. An implementation may, of course, allow the assertion to affect its unspecified behavior, say, by assuming the assertion indicates a condition likely to be true.

6.4 Void assertions

Assertions of type `void` are a feature of [1] but not [2]. They are intended to express requirements that cannot or should not be logically negated with the "`!`" operator. They are useful for expressing requirements that are asymmetric with respect to logical negation, such as:

- a requirement such as `reachable` (relating forward iterators) that is recursively enumerable, but not recursive;
- a requirement such as `in_the_past` (relating a time to a monotonic clock) which is stable, but whose negation is ephemeral;
- a requirement such as `deallocatable` (relating a block of memory to an allocator) whose negation should not be a precondition to any operation; or

- a requirement such as `writable` (for a byte of memory) which, for the sake of efficiency, may be implemented in a weakened or even vacuous form. (A boolean condition cannot be weakened without strengthening its negation.)

The behavior of a void assertion is identical to that of a boolean assertion that cannot return `false`:

```
if ( assertion_testing_specified(ac,tu)
        ? assertions_tested(ac,tu)
        : unspecified() )
   {
    try
       {
        expression;
       }
    catch ( ... )
       {
        fail exceptionally;
       }
   }
```

The behavior of the assertion parameters in a void assertion mirrors the behavior described above for a boolean assertion.

7 Choosing values for the assertion parameters

The five assertion parameters combine to provide eleven possible behaviors for an assertion. Of the eleven, three align with common practice:

To prioritize conservative introduction, set `assertion_testing_specified` to `true`, but `assertions_tested` to `false`. This leaves the specified, defined behavior of a program unchanged by the introduction of assertions.

To prioritize reliable diagnosis, set `assertion_testing_specified`, `assertions_tested`, `assertions_reported`, and `assertions_prevent_continuation` to `true`, and `assertions_assumed` to `false`. This specifies that assertions should be tested and failures reported, and avoids the undefined behavior that may result from continuing on from a failed assertion.

To prioritize speedy execution, set `assertion_testing_specified` to `false` and `assertions_assumed` to `true`. This promotes unspecified and undefined behavior, leaving the greatest scope for optimization. In particular, this combination allows an implementation to formally test the assertion *only in the case the assertion will have undefined behavior, exit with an exception, or produce* `false`. Since such a test leads inevitably to undefined behavior, it need never actually be performed, but the remaining program may be optimized on the assumption that the test would have defined behavior and succeed.

Three other combinations strike balances favoring multiple concerns:

To balance small code size with reliable interruption of control flow, set `assertion_testing_specified` to `true`, `assertions_tested` to `true`, `assertions_assumed` to `false`, `assertions_reported` to `false`, and `assertions_prevent_continuation` to `true`. This combination also provides source obscurity.

To balance reliable diagnosis with conservative introduction, set `assertion_testing_specified` to `true`, `assertions_tested` to `true`, `assertions_assumed` to `false`, `assertions_reported` to `true`, and `assertions_prevent_continuation` to `false`. This allows assertions to be tested and failures to be reported, but does not otherwise affect control flow. Reliable diagnosis is compromised by allowing execution to continue to statements that may rely upon the correctness of the assertion.

To balance speedy execution with conservative introduction, set `assertion_testing_specified` to `false`, `assertions_assumed` to `false`, and `assertions_prevent_continuation` to `true`. This combination gives an implementation the option of testing the assertion in order to optimize later code based on the assertion's success, but does not directly introduce undefined behavior.

The remaining combinations in which testing is left unspecified but success is not assumed allow some opportunistic failure reporting or optimization without introducing undefined behavior. The remaining combinations in which testing is required may be useful in isolating the side effects of the test expressions.

8 Function interfaces

Preconditions and postconditions[6] occupy an awkward space a calling function and the function it calls, which may be in different translation units. This leaves us with a problem: when the assertion parameters differ between the calling and called execution spans, which parameters govern preconditions and postconditions? Or, more concretely, are preconditions and postconditions evaluated as part of the caller or part of the called function?

Every combination of choices has advantages:

- Evaluation as part of the called function centralizes the testing code, leading to smaller code size.
- Evaluation of preconditions in the called function but postconditions in the caller allows each span to verify the conditions it relies upon. A library that chooses `assertions_prevent_continuation` can avoid dangerous behavior using this combination.
- Evaluation of preconditions in the caller and postconditions in the called function places each test in the translation unit where it can most easily be optimized away.
- Evaluation of both preconditions and postconditions in the caller allows a user of a library to insist on testing the interface to the library, even if the library was compiled with no assertion testing. This allows a user to distinguish their own bugs from bugs in the library.

I propose that we split this baby by formally evaluating preconditions and postconditions twice, in both the calling and called execution spans. Specifically, preconditions are to be evaluated first as part of the calling execution span, then as part of the called span; postconditions are to be executed in the called span first.

This proposal, of course, can lead to an unfortunate duplication of assertions. This does not affect the local behavior of a correct program — recall that correct assertions may be repeated any number of times — but can lead to needlessly slow execution or (when `assertions_prevent_continuation` is `false`) duplication of diagnostic messages. I suggest two ways to mitigate this duplication:

First, the assertion categories of preconditions and postconditions in the calling span should be separate from the assertion categories in the called span. That way, a large portion of a project may adopt a uniform policy of testing in the calling or called function, but not both.

Second, when the assertion treatments of the calling and called spans agree, we can allow the duplicate tests to be removed as an optimization. This optimization will, of course, be easiest to apply to calls within a single translation unit.

9 In conclusion

This proposed behavior for assertions is baroque: five translation unit dependent implementation-defined parameters combining to make eleven combinations of implementation-defined, unspecified, and undefined behavior. But I've come to see this complexity as a consequence of the variety of needs listed in section 1.

[6] In relation to [1], I use "precondition" to mean an assertion in a function prologue, and "postcondition" to mean an assertion in an epilogue. Function interfaces in [1] also allow non-assertion statements. For example, a variable may be declared in a prologue and used in the epilogue. A separate implementation parameter may allow the entire interface to go unexecuted, but that is beyond the scope of this paper.

Each operation supports one or more of those needs, and each parameter balances some needs against others. I think it will be difficult to simplify the behavior without disregarding one or more of those needs.

In broad outlines, this treatment of assertions agrees with both [1] and [2]. But I can identify a number of lessons here:

- There is a need for a centralized message reporting mechanism, whose use appears to extend beyond reporting assertion failures.
- Likewise, there is a need for a centralized continuation prevention mechanism, whose use appears to extend beyond escaping from assertion failures.
- Separating the concerns for assertion diagnosis and for preventing continuation in the face of failure provides a flexibility particularly important in contexts that require a small executable size.
- Allowing some assertions to prevent continuation while allowing others to continue after failure promotes use in contexts that require conservative introduction.
- Allowing the treatment of assertions to vary between translation units is useful, but requires some careful wording around implementation-defined behavior.
- We don't have good words for describing behavior that is formally implementation-defined, but with the expectation of user control.
- When the treatment of assertions varies between uncoordinated translation units, either the calling or the called translation unit should be able to require reliable diagnosis of a function interface.
- Library functions explicitly calling for unspecified and undefined behavior are useful in describing the behavior of higher-level language constructs. They may also be useful in code, as they explicitly provide flexibility to the implementation.

While complex, this design does have a pleasing utility: it can vary from a robust diagnostic tool to a zero-cost statement, and potentially to a negative-cost optimization opportunity. The complexity comes from trying to please many constituencies. We may wish to adopt less complex semantics, but I think we should be aware of the constituencies harmed by simplification.

References

[1] L. Lippincott. Procedural function interfaces. Technical Report P0465r0, ISO/IEC JTC1/SC22/WG21, November 2016.

[2] G. Dos Reis, J.D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. Technical Report P0542r0, ISO/IEC JTC1/SC22/WG21, February 2017.

[3] Richard Smith. Working draft, standard for programming language C++. Technical Report N4659, ISO/IEC JTC1/SC22/WG21, April 2017.