Doc No: N4379
Date: 2015-02-08
Author: John Lakos (jlakos@bloomberg.net)
Nathan Myers (nmyers12@bloomberg.net)

FAQ about Contract Assertions

Q: What are contract assertions, in a nutshell?

A: Contract assertions give the programmer a way to deliver precise facts to the toolchain about the expected state of a program at one point.

Q: Aren't contract assertions the way to express all of a function's requirements: its pre- and post-conditions?

A: Contract assertions can sometimes express all of a function's requirements, but often only some of them. Requirements may refer to the program's execution history, which is rarely accessible to a runtime expression. Often requirements can only be expressed in text: “This pointer must indicate a valid memory buffer of N or more bytes”. Contract assertions are for the (important) subset of requirements that actually can be expressed in code.

Q: Are contract assertions meant to support compile-time error checking, dead-code elision, optimization, static analysis, runtime error checking, or runtime error handling? Are they for application developers, library developers, language implementers, or tool developers?

A: Yes! to all of the above. Facts baldly stated offer our tools sharply practical clues as to what we mean. We should not be surprised to find clear expressions of meaning (a.k.a. semantics) broadly useful.

Q: What can contract assertions supported in the core language and library do that can't be done with macros?

A: Contract assertions feed clearly expressed expectations about program state directly to the compiler. The compiler, then, can generate code to verify the expectations at runtime, but it can also determine whether the expectations are

consistent with other facts it has and report inconsistencies as errors, or it can take the expectations as given and tailor optimizations accordingly. Standardizing the runtime-violation reporting mechanism in the library, meanwhile, gives application developers control over the effects of mistakes caught at runtime by contract assertions within the third-party libraries they depend on.

Q: Does N4378, "Language Support for Contract Assertions" compete with other proposals?

A: No. In composing N4378, we were very careful not to preclude choices that might appear in other proposals. N4378 specifies in its WP text a proper subset of what we expect the committee, ultimately, to adopt, so that later proposals will need to mention only what they add.

Q: How does N4378 differ from other proposals?

A: Strictly speaking, there *are* no other proposals, yet. N4378 is the only current proposal we know of that offers a fleshed-out design with WP text. Other papers have discussed features of designs that might be formally proposed someday.

Q: I see that N4378 specifies "compile-time configured assertion levels". What's that about?

A: Assertion levels are akin to the choice whether to compile with "-DNDEBUG=1" as commonly seen in release scripts. However, unlike C assert( ), N4387 contract assertions, when not configured to be checked at runtime, do not disappear. The compiler still sees them, and can use them to catch usage mistakes at compile time, and to produce better object code.

Q: The assertion levels in N4378 (off, min, on, max) don't seem to match my development processes.

A: We chose the levels to be broadly useful. We expect "off" to be used mainly for performance testing, "min" for especially performance-sensitive programs, and "on" for ordinary checks useful in any phase of development, testing, and, usually, production. "Max" is meant to gate disproportionately expensive checks, such as would change the big-O performance of a function, so they are checked only in special circumstances—maybe never. Despite not normally being checked, "max" assertions provide valuable information to the compiler about how a function is used.

Q: Should I expect to be able to link together object files compiled at different assertion levels?

A: Link-compatibility of differently-compiled object files is up to your compiler, just as for the myriad other compiler options you already use. Probably most compilers will allow it, although their choice of which assertions to check and which not may surprise you. In any case, a correct program will behave the same way when compiled at any assertion level.

Q: Some people say they can't use N4378's contract_assert because the assertions do not appear in header files. Why does N4378 allow assertions only in function bodies?

A: The short answer is that anything more ambitious would require defining new syntax that nobody has experience with. N4378 brings us close to our goals with the dead minimum of uncertainty. Wherever assertions are, ultimately, allowed to appear, function bodies will certainly be on the list.

Q: Would allowing contract assertions only in function bodies preclude dead-code elision at call sites? Would it preclude chaining pre- and post-conditions at call sites?

A: Not at all. It would be easy for a compiler to copy contract-assertion expressions it encounters in function bodies into annotations on function symbols in object files, or other descriptive files, making the assertions visible to tools that cannot see into function bodies.

Q: N4378 presents preconditions as the main motivation for contract assertion support. What about postconditions, pre/post-condition chaining, and other use cases?

A: Focusing on preconditions sharpens the presentation, but nobody discounts other use cases. More concretely, the WP text in N4378 does not favor any use over another.

Q: You seem to be suggesting that accepting N4378 will not keep me from getting what I need, even though what I need doesn't seem to be in it.

A: There is plenty we want that isn't in N4378. We mean to propose more later. We might not get everything we want, but it won't be because the other things we want are not on the table yet. N4378 is not the end of a conversation—it is a beginning.

Q: Aside from immediately practical considerations, what should determine whether contract requirements should be expressed in the interface, which is to say in declarations and header files, vs. the implementation, in definitions destined for object files?

A: This is a profound question, with deep consequences.

We could try to place contract requirements according to a distinction between syntax and semantics, so that we place details that affect the way we must write the call to a function in declarations, and details that determine what the function does in the definition. Contract requirements are about semantics, not syntax, so by this criterion they belong squarely in definitions.

Another way would be to place them according to whether they affect actions that happen at compile time, vs. at run time: Compile-time actions in declarations, runtime actions in definitions. This distinction is less useful, because contract assertions have a foot in each.

Or we can place them according to whether they involve types or values, as Stepanov suggests: "*Concepts are requirements on types; preconditions are requirements on values. A concept might indicate that a type of a value is some kind of integer. A precondition might state that a value is a prime number.*" Type-related details (e.g., iterator, integer), then, belong in declarations, and value-related details (e.g. positive, <100, prime) in definitions. By this criterion, again, contract assertions belong squarely in definitions, because contract assertions are all and only about values.

Going back to the definition of a narrow contract, writing some of the requirements as contract assertions in the declaration would seem to be defining, in public, some of what has been specifically documented as undefined. Recall that a big part of why we kept those details undefined was that it left us free to choose what do do in those cases without fear of breaking what clients are allowed to depend on.