Initial Implicit Contract Assertions

Document #:	P3599R0
Date:	2025-01-30
Project:	Programming Language C++
Audience:	SG21 (Contracts), SG23 (Safety and Security), EWG, LEWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <papers@timur.audio>

Abstract

In [P3100R1], we introduced a generic approach to leverage the Contracts facility and to add implicit contract assertions, with the semantics proposed in [P2900R13], to guard against core-language undefined behavior and allow users the flexibility to choose the desired level of runtime checking when building C++ code — with no need for any code changes. In [P3558R0], we proposed that WG21 pursue this approach as a preferred strategy for mitigating undefined behavior at run time and restrict [P3081R1] (Profiles) to static checks.

In this paper, we propose the minimal components needed to introduce implicit contract assertions into the language, along with an initial set of core-language undefined behaviors to guard against with those assertions, adequately covering the runtime checks currently proposed by [P3081R1] while maintaining coherence with the Contracts framework and forward-compatibility with [P3100R1].

Adopting these choices into C++26 will clearly indicate to the world that WG21 is seriously addressing the issues caused by manifestations of undefined behavior in C++ while also maintaining the principles of design and performance that have allowed C++ to become the hugely successful programming language it is today.

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

[P3100R1] introduces the concept of *implicit* contract assertions. Today, we might specify for an operation X that it has *undefined behavior* if some predicate P is not true. With implicit contract assertions, we instead can specify that an operation X includes an implicit contract assertion that P is true. This contract assertion is identical in every way to an *explicit* contract assertion — one that is user-authored using the syntax proposed in [P2900R13] — except that it is inserted automatically by the implementation and no textual representation of it exists within the source code of the C++ program.

Consider, for example, a C++ program that contains within it an access of an array with known bounds:

```
int main(int argc, char *argv[]) {
  int a[10] = { /* ... */ };
  std::size_t i = (argc > 1) ? std::atoi(argv[1]) : 0;
  return a[i];
}
```

When invoked with a command-line argument that is a number greater than or equal to 10, the above program has undefined behavior: Access to an element of `a` that does not exist is attempted.

An implicit contract assertion on array access, however, would change the above program to contain a contract assertion preceding the array access to validate that it is done with an index within the known bounds:

```
int main(int argc, char *argv[]) {
  int a[10] = { /* ... */ };
  std::size_t i = (argc > 1) ? std::atoi(argv[1]) : 0;
  contract_assert( i < (sizeof(a) / size(*a)) );  // inserted by the implementation
  return a[i];
}
```

Just like an explicit contract assertion, an implicit contract assertion can be evaluated with one of the four *evaluation semantics* proposed in [P2900R13]: *enforce*, *quick-enforce*, *ignore*, or *observe*. Depending on how the program is built, this can produce a number of different end results with no further changes to the source code of the program.

- By default, when nothing else is specified, implicit contract assertions on array access will be *enforced*. Executing the program with a command-line argument of `11` will print a diagnostic that an assertion has failed and will terminate the program.
- If built to *quick-enforce* implicit contract assertions on array access, the out-of-bounds access will lead to immediate termination (typically via a trap instruction) without printing a

diagnostic or doing anything else.[1]

- If built to *ignore* implicit contract assertions on array bounds checks, the program behavior will be as it has always been in C++: undefined behavior. Any cases in which the cost of the bounds check truly proves to be significant can be restored to their previous performance levels with this choice.
- If built to *observe* implicit contract assertions on array bounds checks, a diagnostic will be printed, even if followed by undefined behavior, because [P2900R13] specifies that a contract assertion acts as an observable checkpoint (see [P1494R4]). This behavior is even more useful when upgrading an existing project to C++26 since it will allow deploying an existing program without crashing on bugs that were working through happenstance in previous releases. The diagnostics can then be used to asynchronously fix the bugs and improve program correctness without risking production stability.
- When a contract-violation handler is defined by the user, an *enforced* or *observed* implicit contract assertion will invoke that user-defined contract-violation handler with a `kind` of `implicit` and a `detection_mode` of, in this case, `bounds`.

How exactly the evaluation semantic is chosen is, like all other build-time choices, going to be based on command-line arguments provided by our compiler.[2] This approach follows the practice of existing vendor extensions, such as Clang's `-fbounds-safety` or GCC's `-ftrapv`, that introduce implicit runtime checks controlled by compiler flags to mitigate undefined behavior.

This paper is a subset of the more generic framework proposed in [P3100R1] that is actionable in the C++26 timeframe. We propose to introduce implicit preconditions of this form for the following core-language operations.

- A *built-in array subscripting operator*, where the array argument is an array of known bounds `N` and the other argument is an expression whose value is `x`, shall have an implicit precondition that `x >= 0 && x < N`. The value of `std::contracts::detection mode` for these violations will be a new enumerator, `bounds`.
- The *built-in indirection operator* (i.e., the unary `*` operator), whose pointer argument is `p`, shall have an implicit precondition that `p != nullptr`. The detection mode for these violations will be a new enumerator, `dereference`.
- Signed integer arithmetic operations shall have an implicit precondition that they will produce a value in the range of that type.

All of these categories can be extended in the future to more thoroughly cover the undefined behavior that can occur with these operators. Bounds checking can be done on arbitrary arrays or pointers, and indirection that does not result in an object within its lifetime can similarly be checked. In

[1] Diagnostic information about the crash can still be obtained when a debugger is attached or via a symbolicated crash log such as the one produced by Clang's `__builtin_verbose_trap` instruction.

[2] The ability to specify or constrain the evaluation semantic *in code*, as opposed to a build-time option, is provided separately by the *labels* feature proposed [P3400R0]. Section 2.1.5 describes a new directive that can be used to attach user-defined labels (with their corresponding control of evaluation semantic) to categories of implicit contract assertions in a region. Section 2.2.8 describes how those categories can also expose Standard labels with which each category of implicit contract assertions is associated.

both cases, however, it is not *always* possible to evaluate such checks, and for this initial proposal, we are limiting ourselves to contract assertions that can be reliably checked when a user requests them to be.

In the case of integer overflow, in the future we could also introduce a defined fallback behavior (of either wrapping arithmmetic, saturation arithmetic, or producing unspecified erroneous values) in lieu of leaving the overflow itself undefined. Doing so would, however, come with potentially significant performance costs and recouping that performance would require the introduction of the *assume* semantic.

With just these basic tools in the Standard, C++ software of any ilk can be recompiled and then run to prevent major potential software bugs. Combined with library-level checking of contracts (such as precondition assertions on `operator[]` of `std::vector` and `std::span` that can be provided by the Standard Library implementation), significant steps toward both bounds and lifetime safety can be achieved for all C++ software.

2 Existing Proposals

This proposal is a concrete step toward mitigating undefined behavior and thus increasing language safety,[3] but it is also part of a larger context. In particular, it is a subset of a wider framework presented in [P3100R1] that proposes significantly more, in three fundamental ways.

1. A catalog of far more checkable undefined behavior will be produced that can be guarded by implicit contract assertions. With each of these added to the Standard, language safety will be increased, and these additions can be made incrementally.

2. For certain operations, in addition to adding the implicit contract assertion, a new defined fallback behavior is introduced. This allows the *ignore* semantic to be used while still limiting the potential downsides, though it may come at a performance cost, effectively making the previously *undefined behavior* into *erroneous behavior*.

3. To recoup the potential performance cost of defining the undefined behavior, a new *assume* semantic is introduced that restores the same behavior and performance characteristics the code would have if compiled today.

Importantly, none of the above steps are necessary to start introducing implicit contract assertions into the language, as we are doing in this paper. Each step can be taken incrementally on top of this proposal, and each will add to the language safety and power of C++ software as the Standard evolves. Note that each of the above features can be incrementally adopted in a future Standard and yet still apply — because the behavior today is undefined — to C++ code compiled with earlier Standards. We need not rush introducing these facilities before we are confident in their effectiveness or appropriateness, yet they will benefit engineers far more quickly than typical changes introduced into the C++ draft Standard once consensus is achieved to move forward with each one.

[3] Throughout this paper, we avoid using the ambiguous term "safety" because distinguishing between *language safety* (i.e., the absence of unbounded undefined behavior) and *functional safety* (i.e., the avoidance of unintended harm to humans, property, or the environment) is critically important. We recommend that all other papers in this design space follow this practice; see [P3500R0] and [P3578R0] for a more thorough discussion.

Further, this proposal has overlap with [P3081R1]. That paper introduces a number of core safety profiles. Two of these profiles add implicit contract assertions when enabled, using the terminology that certain conditions are *profile-checked*.

1. The *bounds* profile profile-checks that subscripting operators are within bounds by asserting `std::index_in_range` with appropriate arguments.
2. The *lifetime* profile profile-checks that the dereferencing operator is not evaluated on an operand of type `P` whose value is equal to `P{}`.

For user-defined types, the far better and never-wrong solution is to use explicit contract assertions — `pre` — to check the actual requirements of the types involved for using them within contract. As a general rule, implicit contract assertions apply to core-language operations for which the contract is fully known to the compiler, while explicit contract assertions should be used for user-defined code since the contract of such code is known only to the human developer.

For built-in types, this paper proposes implicit contract assertions for the same cases covered by [P3081R1]; however, as proposed here, they are present in *all* C++ programs and without any need to opt-in to having a profile enabled in either the source code or on the compiler command line. We recommend that [P3081R1] should be restricted to *static* (compile-time) checks; *runtime* checks (i.e., implicit contract assertions) should instead be adopted as proposed here (a more thorough discussion can be found in [P3558R0]) for both a safer default and a design direction that is forward-compatible with the wider application of this basic design proposed in [P3100R1].

Finally, the definition of *implicit contract assertions* proposed here is identical to the one proposed in [P3229R0], which does not deal with undefined behavior but is a different subset of [P3100R1] also targeting C++26.

3 Wording Changes

The proposed wording changes are relative to [P2900R13] and [N5001]. Wording introducing the concept of *implicit contract assertions* is consistent with that proposed in [P3229R0].

Modify [expr.pre] paragraph 4 as follows:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type `T`, the behavior is undefined. If `T` is a signed integral type, an implicit precondition assertion ([basic.contracts.implicit]) that the result is in the range of representable values for `T` is applied to the operation. [*Note:* Treatment of division by zero, forming a remainder using a zero divisor, and all floating-point exceptions varies among machines, and is sometimes adjustable by a library function. — *end note*]

Modify [expr.sub] paragraph 2 as follows:

> [2] With the built-in subscript operator, an *expression-list* shall be present, consisting of a single *assignment-expression*. One of the expressions shall be a glvalue of type "array of `T`" or a prvalue of type "pointer to `T`" and the other shall be a prvalue of unscoped enumeration or integral type. The result is of type "`T`". The type "`T`" shall be a completely-defined object type. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`,

except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise. If the type of either expression is "array of `N T`", and the other operand is `I`, an implicit precondition assertion ([basic.contracts.implicit]) that `0 <= I && I < N` is applied to the operation.

Modify [expr.unary.op] paragraph 1 as follows:

1 The unary `*` operator performs indirection. Its operand shall be a prvalue of type "pointer to `T`", where `T` is an object or function type. The operator yields an lvalue of type `T`. If the operand `p` points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in [expr.typeid]. An implicit precondition assertion ([basic.contracts.implicit]), that `p != nullptr`, is applied to the operation.

Modify [basic.contract.general] starting at paragraph 1 as follows:

1 Contract assertions ~~allow the programmer to~~ specify properties of the state of the program that are expected to hold at certain points during execution. *Explicit* c~~C~~ontract assertions are introduced by *precondition-specifier*s, *postcondition-specifier*s ([dcl.contract.func]), and *assertion-statement*s ([stmt.contract.assert]). *Implicit* contract assertions are applied to operations by the implementation.

2 Each contract assertion has a *predicate*, which is an expression of type `bool`. [*Note:* ~~The value of the predicate is used to identify program states that are expected.~~ If it is determined during program execution that the predicate has a value other than `true`, a *contract violation* occurs. A contract violation is always the consequence of incorrect program code. — *end note*]

Modify [basic.contract.eval] starting at paragraph 1 as follows:

1 An evaluation of a contract assertion uses one of the following four *evaluation semantics*: *ignore*, *observe*, *enforce*, or *quick-enforce*. Observe, enforce, and quick-enforce are checking semantics; enforce and quick-enforce are terminating semantics.

2 Which evaluation semantic is used for any given evaluation of a contract assertion is implementation-defined. During constant evaluation, implicit contract assertions are always evaluated with the enforce semantic. [*Note:* The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

Add a new section, [basic.contract.implicit] after [basic.contract.eval]:

1 A built-in operation O may have an *implicit precondition assertion* C applied to it. If so, the evaluation of C is sequenced before the evaluation of O and after the evaluation of all operands of O.

2 A built-in operation O may have an *implicit postcondition assertion* C applied to it. If so, the evaluation of C is sequenced after the evaluation of O.

Modify [support.contracts.enum.kind] paragraph 1:

[1] The enumerators of `assertion_kind` correspond to the possible syntactic forms of a contract assertion ([basic.contract]):

- `assertion_kind::implicit`: the evaluated contract assertion was an implicit contract assertion.
- `assertion_kind::pre`: the evaluated contract assertion was a precondition assertion.
- `assertion_kind::post`: the evaluated contract assertion was a postcondition assertion.
- `assertion_kind::assert`: the evaluated contract assertion was a assertion-statement.

Modify [support.contracts.enum.detection] paragraph 1:

[1] The enumerators of `detection_mode` correspond to the manners in which a contract violation ([basic.contract.eval]) can be identified:

- `detection_mode::predicate_false`: the contract violation occurred because the predicate evaluated to `false` or would have evaluated to `false`.
- `detection_mode::evaluation_exception`: the contract violation occurred because the evaluation of the predicate exited via an exception.
- `detection_mode::bounds`: the contract violation occurred due to an out-of-bounds access into an array ([expr.sub]).
- `detection_mode::dereference`: the contract violation occurred due to an attempt to dereference a null pointer ([expr.unary.op]).
- `detection_mode::arithmetic`: the contract violation occurred due to an integer operation whose result would not be representable ([expr.pre])

Acknowledgments

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[N5001] Thomas Köppe, "Working Draft, Programming Languages – C++", 2024
http://wg21.link/N5001

[P1494R4] S. Davis Herring, "Partial program correctness", 2024
http://wg21.link/P1494R4

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2025
http://wg21.link/P2900R13

[P3081R1] Herb Sutter, "Core safety profiles for C++26", 2025
http://wg21.link/P3081R1

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne, “Undefined and erroneous behaviour are contract violations”, 2024
http://wg21.link/P3100R1

[P3229R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Making erroneous behaviour compatible with Contracts”, 2025
http://wg21.link/P3229R0

[P3400R0] Joshua Berne, “Specifying Contract Assertion Properties with Labels”, 2025
http://wg21.link/P3400R0

[P3500R0] Timur Doumler, Gašper Ažman, and Joshua Berne, “Are Contracts "safe"?”, 2025
http://wg21.link/P3500R0

[P3558R0] Joshua Berne and John Lakos, “Core Language Contracts By Default”, 2025
http://wg21.link/P3558R0

[P3578R0] Ryan McDougall, “Language Safety and Grandma Safety”, 2025
http://wg21.link/P3578R0