Proposing Contract Attributes

Document #: WG21 N4435
Date: 2015-04-09
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper explores the use of C++ attributes as a vehicle to express the traditional components of contract programming: preconditions, postconditions, and invariants. We evaluate the applicability of attributes for this purpose, and present for EWG discussion a strawman syntax and semantics for three new attributes.

1 Introduction

In the last decade, WG21 has reviewed a significant number of papers on the general subject of C++ programming with contracts (preconditions, postconditions, invariants). Some of these papers (e.g., [N4110]) have explored the design space, some (e.g., [N1613], [N4154], and [N4248]), have proposed new or repurposed keywords, some (e.g., [N4319]) have provided reports of prior art in other languages and/or libraries, and some (e.g., [N4135]) propose library solutions.[1]

However, none of these papers has to date explored the relationship of C++ attributes to contracts.[2] We will do so in the remainder of this paper, recommending that EWG evaluate this contribution as a proposed direction for future work. Although most previous proposals focus only on function preconditions, we will address preconditions, postconditions, and several forms of invariants.

First, we briefly explore how contracts satisfy our accepted guidelines for attributes.

2 Contracts vis-à-vis attributes

In §7 of [N2761], Maurer and Wong address the question, "what should be an attribute and what should be part of the language." They conclude that "attributes should not affect the type system, and not change the meaning of a program regardless of whether the attribute is there or not. Attributes provide a way to give [a] hint to the compiler, or can be used to drive out additional compiler messages that are attached to the type, or statement."

[1] See this paper's Bibliography (§7) for references to earlier or later versions of some of the papers cited here.

[2] See this paper's Addendum (§6) for a recent development.

We believe that contract assertions are a good fit to this guideline. As they do not affect a program's semantics, such attributes are freely ignorable by a compiler. When not ignored, a compiler can use the attribute contracts' information as hints in order to achieve better diagnostics and/or more robust and efficient code. More specifically, such attributes can be optionally used by compilers to achieve each of the benefits Dos Reis set forth in [N4319], namely:

- "Runtime checks, complement to static type checking, for early containment of undesired program behavior
- "Support for testing
- "Executable documentation of functions, and of data structure invariants
- "Optimizations (of expensive runtime checks) enabled by statically provable contracts
- "Compile-time detection and elimination of bugs through static analysis"

3 Preconditions and postconditions

We begin with a proposal for preconditions. Since compile-time preconditions are already available via the **`requires`** clause in the Concepts Lite Technical Specification [N4377], we will focus on run-time preconditions.

We envision a new attribute, **`pre:`**, that can appertain to any function, function template, member function, or member function template declaration:

```
template< class FwdIterator, class T >
bool
  std::binary_search( FwdIterator first, FwdIterator last, T const& value)
  [[ pre:(std::is_sorted(first, last)) ]];
```

(Yes, the attributes are positioned in a non-canonical location within the declaration, but we want the function parameters' names to be in scope.) Syntactically, the precondition is expressed as an expression of a type convertible to **`bool`**, positioned immediately after the **`pre:`** introducing the attribute. We require the attribute's predicate to be pure (i.e., free of side effects), else the behavior is undefined.

Given such an annotation, we propose to leave it as implementation-defined behavior as to what to do with the expression. For example, the implementation may generate code to evaluate the expression at call sites, or not. If such code is generated, the implementation may, for example, choose to generate code that throws an exeception if precondition evaluation proves **`false`**.

As another possibility, the implementation may be able to deduce, at compile time, that a precondition's predicate will always hold, and therefore that it need never be evaluated at a particular call site. This could arise, for example, by observing that an earlier postcondition implies that the later precondition holds.

```
template< class RandomAccessIterator >
void
  std::sort( RandomAccessIterator first, RandomAccessIterator last );
  [[ post:(std::is_sorted(first, last)) ]];
```

We similarly envision a new attribute, **`post:`**, that can appertain wherever a **`pre:`** attribute appertains. A postcondition is not expected to hold if the called function either fails to return or returns via an exception, **`longjmp`**, etc. If a postcondition's predicate is to be evaluated, it

must be evaluated only after a function has returned control to the calling program via a **return** statement or equivalent.

4 Invariants

We envision an invariant attribute, **inv:**, that can appertain to functions, classes, looping constructs, variables, etc.

When appertaining to a function (or member function, etc.), an invariant serves as a precondition and as a postcondition.

```
1 template< class FwdIterator, class T >
2 bool
3   std::binary_search( FwdIterator first, FwdIterator last, T const& value)
4   [[ inv:(std::is_sorted(first, last)) ]];
```

If a function has a precondition as well as an invariant attribute, the conjunction of their predicates is the function's complete precondition. Similarly, if a function has a postcondition as well as an invariant attribute, the conjunction of their predicates is the function's complete postcondition.

With two exceptions, when an invariant appertains to a class type, the invariant implicitly appertains to each **public** and **protected** member function. The exceptions are for constructors and destructors: the class invariant supplies only a postcondition for a constructor, and supplies only a precondition for a destructor. This is because a class invariant is initially established when a constructor completes execution, and is no longer applicable once a destructor begins execution. Additional pre- and post-conditions are permitted for constructors and destructors, as for any member function, on a per-function basis. However, the efficacy of a class invariant is proposed to match the lifetime of an instance of the class.

Because it seems useful for class member names to be in scope when formulating a class invariant, we would propose to allow attributes for class invariants to follow the class definition, just as we prefer functions' pre- and postconditions to follow a function's declaration. (It may also be useful to permit **this** to appear in a class invariant, but such an option needs further study.)

When an invariant appertains to a **while** or **for** statement, and the compiler chooses to evaluate the associated predicate, it does so each time and immediately before the loop's predicate is evaluated. When an invariant appertains to a **do** statement, and the compiler chooses to evaluate the invariant's predicate, it does so on each iteration immediately before the loop body is executed.

When an invariant appertains to a variable or to a data member, and the compiler chooses to evaluate the invariant's predicate, it does so immediately after the variable is constructed, and immediately after each subsequent update to that variable.

If the compiler chooses to do so, predicates of invariants not described above are evaluated *in situ*. Such invariants are often used to describe and reason about the progress made so far toward the program's desired final state.

5 Acknowledgments

We gratefully acknowledge, with many thanks, the thoughtful remarks received from the readers of this paper's early drafts.

6 Addendum

Since this paper was drafted, we have been made aware of "Simple Contracts for C++ (Draft)," an as-yet unpublished paper[3] that proposes functionality overlapping significantly with the pre- and post-condition portion (§3) of our paper. We support that shared objective, of course, but believe there should be a discussion (probably in EWG) regarding WG21's ultimate contract programming goals. To that end, our paper contributes a menu of options that we consider feasible.

7 Bibliography

[N1613] Thorsten Ottosen: "Proposal to add Design by Contract to C++." ISO/IEC JTC1/SC22/WG21 document N1613 (post-Sydney mailing), 2004-03-29. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1613.htm.

[N1669] Thorsten Ottosen: "Proposal to add Design by Contract to C++ (revision 1)." ISO/IEC JTC1/SC22/WG21 document N1669 (pre-Redmond mailing), 2004-09-10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1669.htm.

[N1773] David Abrahams, Lawrence Crowl, Thorsten Ottosen, and James Widman: "Proposal to add Contract Programming to C++ (revision 2)." ISO/IEC JTC1/SC22/WG21 document N1773 (pre-Lillehammer mailing), 2005-03-04. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1773.htm.

[N1800] Lawrence Crowl and Thorsten Ottosen: "Contract Programming for C++0x." ISO/IEC JTC1/SC22/WG21 document N1800 (post-Lillehammer mailing), 2005-04-27. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1800.htm.

[N1866] Lawrence Crowl and Thorsten Ottosen: "Proposal to add Contract Programming to C++ (revision 3)." ISO/IEC JTC1/SC22/WG21 document N1866 (pre-Tremblant mailing), 2005-08-24. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1866.htm.

[N1962] Lawrence Crowl and Thorsten Ottosen: "Proposal to add Contract Programming to C++ (revision 4)." ISO/IEC JTC1/SC22/WG21 document N1962 (pre-Berlin mailing), 2006-02-25. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.htm.

[N2761] Jens Maurer and Michael Wong: "Towards support for attributes in C++ (Revision 6)." ISO/IEC JTC1/SC22/WG21 document N2761 (post-San Francisco mailing), 2008-09-18. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.htm.

[N3604] John Lakos and Alexei Zakharov: "Centralized Defensive-Programming Support for Narrow Contracts." ISO/IEC JTC1/SC22/WG21 document N3604 (pre-Bristol mailing), 2013-03-18. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3604.htm.

[N3753] John Lakos and Alexei Zakharov: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 1)." ISO/IEC JTC1/SC22/WG21 document N3753 (pre-Chicago mailing), 2013-08-30. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3753.htm.

[N3818] John Lakos and Alexei Zakharov: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 2)." ISO/IEC JTC1/SC22/WG21 document N3818 (post-Chicago mailing), 2013-10-11. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3818.htm.

[N3877] John Lakos and Alexei Zakharov: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 3)." ISO/IEC JTC1/SC22/WG21 document N3877 (pre-Issaquah mailing), 2014-01-17. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3877.htm.

[N3963] John Lakos and Alexei Zakharov: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 4)." ISO/IEC JTC1/SC22/WG21 document N3963 (post-Issaquah mailing), 2014-03-02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3963.htm.

[3] Authors: Gabriel Dos Reis, J. Daniel García, Francesco Logozzo, Manuel Fähndrich, and Shuvendu Lahiri. Attachment to posting [c++std-ext-16696].

[N3997] John Lakos, Alexei Zakharov, and Alexander Beels: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 5)." ISO/IEC JTC1/SC22/WG21 document N3997 (pre-Rapperswil mailing), 2014-05-27. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3997.htm.

[N4075] John Lakos, Alexei Zakharov, and Alexander Beels: "Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)." ISO/IEC JTC1/SC22/WG21 document N4075 (post-Rapperswil mailing), 2014-06-20. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4075.htm.

[N4110] J. Daniel Garcia: "Exploring the design space of contract specifications for C++." ISO/IEC JTC1/SC22/WG21 document N4110 (post-Rapperswil mailing), 2014-07-06. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4110.htm.

[N4135] John Lakos, Alexei Zakharov, Alexander Beels, and Nathan Myers: "Language Support for Runtime Contract Validation (Revision 8)." ISO/IEC JTC1/SC22/WG21 document N4135 (pre-Urbana mailing), 2014-10-09. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4135.htm.

[N4154] David Krauss: "Operator **`assert`**." ISO/IEC JTC1/SC22/WG21 document N4154 (pre-Urbana mailing), 2014-10-04. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4154.htm.

[N4248] Alisdair Meredith: "Library Preconditions are a Language Feature." ISO/IEC JTC1/SC22/WG21 document N4248 (pre-Urbana mailing), 2014-10-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4248.htm.

[N4319] Gabriel Dos Reis, Shuvendu Lahiri, Francesco Logozzo, Thomas Ball, and Jared Parsons: "Contracts for C++: What Are the Choices?" ISO/IEC JTC1/SC22/WG21 document N4319 (post-Urbana mailing), 2014-11-23. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4319.htm.

[N4377] Andrew Sutton: "Programming Languages—C++ Extensions for Concepts." ISO/IEC JTC1/SC22/WG21 document N4377 (mid-Urbana/Lexena mailing), 2015-02-09. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf.

8 Document history

Version	Date	Changes
1	2015-04-09	• Published as N4435.