N4415 2015-04-12 Reply-To: gdr@microsoft.com

Simple Contracts for C++

Gabriel Dos Reis
Microsoft

J. Daniel García
Universidad Carlos III de Madrid

Francesco Logozzo
Facebook

Manuel Fähndrich
Google

Shuvendu Lahiri
Microsoft

Abstract

We present a minimal system for expressing interface requirements as contracts. They provide basic mitigation measures for early containment of undesired program behavior. The set of facilities suggested in this proposal is deliberately kept to the minimum of pre-conditions and post-conditions. Contracts are part of an operation's interface, but not part of its type. That is, while the expression of contracts is logically part of the operation's interface, the actual code verifying the conditions are part of the operation's implementation.

1 WHAT ARE CONTRACTS?

Contracts are requirements that an operation puts on its arguments for successful completion and set of guarantees it provides upon successful completion. The former is known as *pre-condition* and the latter is called *post-condition*. Contracts are part of the interface of an operation as a programmer (consumer or producer) sees it, e.g. "What do I have to do to call this function?" and "What may I rely on when implementing this function?" However, contracts (as suggested in this proposal) are not part of the type system.

Contracts are not a general error reporting mechanism, nor are they substitute for testing frameworks. Rather, they offer a basic mitigation measure when a program goes wrong because of mismatch of expectations between parts of a program. Contracts are conceptually more like structured `assert()` integrated into the language, playing by the language semantics rules – therefore basis for principled program analysis and tooling.

There is a strong desire in the C++ community, as evidenced by the number of contracts proposals [1] [2] [3] [4] [5] [6] and continued discussions in the C++ committee (e.g. at the Urbana, IL, 2014 meeting) for language support to express contracts directly in a program.

1.1 CONTRACTS: INTERFACE SPECIFICATION

Contracts are part of the interface of an operation, but not part of its type. A contract expresses what the caller of a function must do to satisfy the expectations the callee places on its arguments. Consequently, **the expression of a contract must logically be part of the declaration of the operation**.

1.2 CONTRACTS: CHECKING IS IMPLEMENTATION

For simplicity and usability reasons, contracts are not part of the type of an operation. However, ideally, a function invocation should display the same observable behavior whether the function is called directly (e.g. by name) or indirectly (e.g. via a pointer to function). Therefore, **the checking of contract, the concrete codes asserting the requirements or the promise an operation makes, must conceptually be part of the implementation of the operation**.

1.3 SYNTAX

So, how we do express contracts in code? Clearly we need a syntactic place to put a pre-condition or a post-condition. There are various ways to achieve this. Either a contract is expressed in the declaration of a function, or it is expressed separately through a "proclamation" declaration of some sort. The one concern with expressing contracts in declaration is that the "obvious" place to put contracts is becoming "crowded". An advantage of specifying a contract in a proclamation declaration is that it could be done "retroactively, after the fact" separate from function declaration and does not need to compete for the syntactically crowded space. However, this is also a problem as one must now maintain coherence between declarations, definitions, and proclamations. Furthermore, member functions can be declared only once and a general design principle of C++ is that all that is to be known about a class "interface" is known at the closing brace of its definition. Therefore, contracts must appear in the function declaration – not as a post facto extension.

Should we reach out for new keywords, e.g. **expects** for pre-conditions and **ensures** for post-conditions? This proposal takes an alternative route and suggests the use of C++ attributes for expressing contracts:

- `[[expects: condition]]` for saying that an operation expects `condition` to hold for a call to complete successfully
- `[[ensures: condition]]` for saying that an operation guarantees `condition` to hold after a successful call

For example, a pre-condition contract of the indexing operator of a `Vector` class could be written:

```
T& operator[](size_t i) [[expects: i < size()]];
```

Similarly, a post-condition contract on a constructor of an `ArrayView` class could be expressed as:

```
ArrayView(const vector<T>& v) [[ensures: data() == v.data()]];
```

Note that in a correct program, contracts can be freely ignored without changing the observable behavior. This is in line with the general understanding of semantics impacts of attributes on correct programs.

1.4 OPERATIONAL SEMANTICS

The simplest operational semantics of a contract is as follows:

- The "condition" of a contract (pre-condition or post-condition) is type-checked in the scope of the function's parameter declarations. For a member function, that includes the enclosing class's scope. The same holds for any (friend) function lexically declared in a class.

- The pre-condition of an operation is evaluated before any other statement in the function's body. If the result is true, then normal control of execution continues to the first statement in the body of the function. Otherwise, further execution is not guaranteed: either the program aborts, or throws an exception, or if it is allowed to continue then the behavior is undefined. Whichever of these alternatives is chosen is implementation-defined. That is, an implementation may offer translation modes to check all contracts, or only pre-conditions, or only post-conditions, or ignore contract checking.
- Similarly, the post-condition is evaluated after evaluation of the return value (if any) and after the destruction of any local variables, but before control is transferred back to the caller. If the evaluation yields true then control continues as it normally would. Otherwise, the program is abnormally terminated as in the pre-condition case. Note that the post-condition of a function is not evaluated as part of an exceptional transfer of control.

1.5 Control of Contract Assertions

As explained in the previous section, the basic conceptual model of pre-condition or a post-condition is as if the expression `assert(condition)` is evaluated at the appropriate place. There are several design choices and practical considerations here.

First, there ought to be an ability to turn on and off contract checking, or just to enable partial contract checking (e.g. only pre-conditions, or only post-conditions). We do not believe that this facility has to be in form of "feature test macros" accessible in the source program. Remember that for a correct program executed with correct data, ignoring contracts (e.g. turning contract checking off) should not have any effect on permissible observable behavior of the program. It is in some sense a form of optimization – dead code removal. Consequently, we encourage implementations to offer switches to select level of contract checking: on, off, pre-condition only, post-condition only.

Second is the question of the granularity of control. Should contract checking be control per function declaration basis, per class definition, per namespaces, per translation, or just whole sale program? Clearly, a per-function or whole-program control is impractical for most programs. Similarly, a per-class or per-namespace control is a road to anarchy. This proposal suggests a per-translation unit control of contract assertion.

Finally, an `std::abort()` in case of contract failure may not be appropriate for some programs – despite the fact that today, a contract failure results in undefined behavior; at least as far as standard library components are concerned. For programs that can afford it or need it, it might makes sense for implementations to offer throwing exceptions (such as `std::precondition_failure`, `std::postcondition_failure`) instead of an unrecoverable program termination via `std:abort()`. However, it is a critical design criteria that contracts be usable in embedded systems or other resource-constrained systems that cannot afford exceptions. A callback mechanism with setting of various pointers to functions to control contract assertion is equally challenging in terms of removing "dead codes" – many safety-critical systems operate under strict policies of not including codes they don't run.

Consistent with EWG's expressed preference at the 2014 Urbana, IL, meeting, we recommend that the means of contract assertions be implementation-defined, but should allow "all, none, pre-condition, post-condition" contract checking on per translation unit basis.

1.6 Facilities Deliberately Left Out

This proposal has an extreme focus on simplicity and deliberately leaves out several facilities found useful in more elaborate "programming by contracts" systems. These include "invariants", "abstract states", ability to reference "old" values of a parameter upon entry in function-body, reference to the return value of a function in a post-condition, conditions on exceptional transfer of control, etc. These facilities are left out *not* as a result of value judgment about their usefulness. Rather, we put a premium on simplicity and an evolutionary approach to contracts for C++.

Note that there are at least two notions of invariants: (1) representation invariants; and (2) logical invariants. A representation invariant is generally about the object representation of a class, whereas a logical invariant expresses invariants about the abstract data structure that a class is designed to materialize. For example, take a `RedBlackTree` class designed to represent a red black tree. Assume further that a `RedBlackTree` object contains the root node as member. A representation invariant expresses a constraint on that root node (i.e. the direct member of the tree object) that it is colored "black", where as a logical invariant may express the fact every node reachable from the root and that is colored "red" has two children colored "black" and that every path from a given node to its descendant leaves contains exactly the same nodes colored "black". This example shows a similarity with the 'physical const' vs. 'logical const' distinction in current C++. Consequently, we are postponing invariants as possible future extensions of this minimal contract system.

1.7 What Are the ABI Impacts?

Does this proposal require an ABI change or an ABI breakage? No. This proposal does not break ABI, nor does it require an ABI change or innovation. An implementation that systematically ignores contracts after type checking, is a conforming implementation. Similarly, an implementation that systematically inserts `assert()` corresponding to pre-conditions and post-conditions in function bodies is also a conforming implementation. None of these implementation strategies requires ABI modification or invention. Similarly, anything in between (e.g. checking pre-conditions only, or checking post-conditions only) does not require an ABI change.

On the other hand, an implementation can take advantage of the additional information available in contracts for code generation purposes, as long as it satisfies the usual "as if" rule. In particular, an implementation with multiple entrypoint/exitpoint features may push contract checking to call sites if judged beneficial. However, none of this is required by this proposal.

2 Language Integration

2.1 Multiple Declarations

When a function can be declared multiple times (e.g. at namespace scopes), should contracts be repeated or omitted? Well, ideally, an entity should be declared only once – the default practice in the module world. However, for simplicity, we suggest the following: in a given translation unit, if a declaration of an entity has a contract, then any subsequent declaration that mentions the same contract should have identical expression (in the ODR sense) of that contract. Furthermore if a function declaration has a contract, then its definition must also have that contract. We do not require implementations to check

this rule across translation units. Note that it is permitted for a declaration not to have a contract and only the definition to mention one. This allows programs to conceal contracts from public interfaces, as questionable as that might be.

2.2 Contract Conditions and Side Effects

What kind of expression is acceptable in contracts? Contract conditions should be side effect free. That is, their evaluation should not produce any difference in the program's observable behavior. One could approximate this requirement by saying that contract conditions are as if they were the body of a constexpr function. However, we don't expect most useful contract conditions to involve only constexpr functions in practice (e.g. std::vector<T>::size). One could also attempt to define yet another class of expressions and require implementations to enforce those restriction s, but that is added complexity with little benefit. We've concluded that it is more effective to simply state that contract conditions are expected to be side effect free.

2.3 Virtual Functions Overrider

A virtual function overrider inherits the contracts from the base class function it is overriding. However, if an overrider repeats a contract, it must match exactly the original function declaration. It cannot weaken nor can it strengthen the contract. Note that this restriction also applies to a virtual function that simultaneously overrides a function from several base classes. Example:

```
struct A {
  bool f() const;
  bool g() const;
  virtual string bhar() [[expects: f() && g()]];
  virtual int hash() [[ensures: g()]];
  virtual void gash() [[expects: g()]];
  virtual double fash(int i) const [[expects: i > 0]];
};
struct B : A {
  string bhar() override [[expects: f()]];      // ERROR: weakening.
  int hash() override [[ensures: f() && g()]];  // ERROR: strengthening.
  void gash() override [[expects: g()]];        // OK: repeat from base.
  double fash(int) override;                 // OK: inherited from base.
};
```

Note that weakening a pre-condition of an overrider is technically sound; this proposal does not suggest that ability for simplicity reasons. Similarly, strengthening a post-condition is theoretically sound; this proposal does not propose that capability out of simplicity concerns.

2.4 Accessibility of Members Referenced In Contracts

Since a contract is part of the interface of a function and class members can be referenced in the expression of a member function contract, there have been some concerns over possible abstraction leakage. We propose a very simple rule: class members referenced in a contract for a member function should be of an accessibility at least as permissive as the member function itself. That is:

- A public member function can only reference public members in its contracts

- A protected member function can reference protected or public members in its contracts
- Finally, a private member function can reference all members in its contracts.

To close the loop, a friend declaration of a function lexically at a class scope can only reference public members of that class.

2.5 Attribute Syntax

The attribute syntax for contracts suggested in this proposal, e.g. `[[expects: condition]]` or `[[ensures: condition]]`, does not strictly conform to the C++11 notation. For that, it would have to use a matching pair of parenthesis around the "condition" instead of a colon: `[[expects(condition)]]` and `[[ensures(condition)]]`. We find that in practice, the colon notation (as suggested in this proposal) makes for easier to read contracts than the more lispy C++11 attribute notation. This small extension to attribute syntax is useful (at least for readability) beyond contracts.

2.6 Function Pointers

Contracts are not part of the type system. In particular, the address of a function with contracts has the function type as if there were no contract. Example:

```
double f(double x) [[expects: x >= 0]];
double (*pf)(double) = &f;       // OK.
```

However, it is possible to declare a pointer to a function with a contract. Initializing or assigning to such pointer is valid only if the contracts are equal. Example:

```
double f(double x) [[expects: x >= 0]];
double (*pf)(double x) [[expects: x >= 0]] = &f;     // OK.

double g(double);
double (*pg)(double x) [[expects: x != 0]] = &g;     // ERROR.
```

For the same reasons, if function type (or a pointer to function type) alias is used to declare a function, any contracts in the alias declaration does not transfer to the function or to the pointer to function. This is just a particular instance of a much more general problem that implementations face today, outside of any contract considerations. Consequently, any "solution" in this space should be applicable to non-contract declarations. Any "solution" in this space will have to deal with (type) template argument deduction.

3 Synergy with Analysis Tools

As reported previously [7], production analysis tools for C++ programs can build on standard source-level contract annotations to provide greater reliability and safety. Furthermore, many popular C++ bug finding tools support custom annotations to allow programmers to express intents to add checks and suppress false alarms. In particular:

- Clang, quote from [8]: "The Clang frontend supports several source-level annotations in the form of GCC-style attributes and pragmas that can help make using the Clang Static Analyzer more useful.

These annotations can both help suppress false positives as well as enhance the analyzer's ability to find bugs."

- SAL, quote from [9]: "SAL is the Microsoft source code annotation language. By using source code annotations, you can make the intent behind your code explicit. These annotations also enable automated static analysis tools to analyze your code more accurately, with significantly fewer false positives and false negatives."
- The popular Coverity static analysis tools for C/C++ provide source code annotations to suppress false positives [10].

A standard notation for expressing interface requirements will help reduce the fragmentation of the analysis tools ecosystem while fostering portable checking.

4 COMPARISON TO BLOOMBERG'S PROPOSAL: N4378

The latest Bloomberg proposal (10th iteration) [11] is a scaled down version of previous iterations [3] [6]. Although it is titled "Language Support for Contract Assertions", the language support is not apparent and the system is best viewed (and most effectively used) as "an assertion framework". The entire infrastructure relies on manual insertion of assertions at points where the programmer intends to have a check in an implementation. There is no provision for expressing contracts at the interface level. Nor is there a formal structure that compilers and analysis tools can effectively use, and at scale. We suspect the intent here is that contracts are best expressed in informal English, never in code at the interface level, and checks should be inserted in implementation. Obviously, this does not help analysis tools, nor does it help ensure that actual contracts are expressed unambiguously to callers. Furthermore, the global nature of the assertion control callback is probably suitable for development environments with a central authority that controls how assertions are used; but it is not suitable for most environments where programs are composed out of several parts possibly developed by different teams or organizations or components developed with different constraints. We should not underestimate the value of per-component control of contract assertions.

Perhaps, the most obvious point of synergy is that the "cheap to evaluate assertions" of N4378 that are placed at the beginning of a function definition may be candidates as formal pre-conditions (as defined in this proposal).

5 COMPARISON TO N4293

The proposal N4293 [12] by the second author is a revision based on committee discussions held at the 2014 Urbana, IL, meeting, which also includes a summary of EWG directions.

The most important difference between this proposal and N4293 is syntax. While N4293 adds new keywords for contracts, here we propose to use attributes. As explained in earlier sections, this choice underscores the notion that removing contracts from a correct program does not change its observable behavior. This is in complete alignment with the general expected use of attributes. Furthermore, N4293 required checking the absence of side effects in contracts while this proposal does not require this kind of checks.

There are several suggested facilities from N4293 that are not provided by this proposal, for the sake of simplicity. They can be considered for future extensions. These features include references to previous parameters values, and reference to a function return value in post-conditions. Besides, in N4293 several checking modes where defined. In contrast, this proposal leaves freedom to implementations on several valid approaches for correct programs.

6 FORMAL WORDING

The following proposed modifications to the language definition are with respect to the Working Draft, document number N4296.

Augment the grammar production *attribute* in paragraph 7.6.1/1 as follows

> *attribute*:
> *attribute-token attribute-argument-clause_opt*
> *attribute-token* **:** *balanced-token-seq*

Modify paragraph 7.6.2/3 as follows:

> […] Unless specified otherwise, if a keyword (2.11) or an alternative token (2.5) that satisfies the syntactic requirements of an *identifier* (2.10) is contained in an *attribute-token*, it is considered an identifier. Unless specified otherwise, no name lookup (3.4) is performed on any of the identifiers contained in an *attribute-token*.

Add a new section 7.6.6 titled "Contracts" as follows:

> An *attribute-specifier* of the form `[[expects:` *balanced-token-seq*`]]` or `[[ensures:` *balanced-token-seq*`]]` is a *contract*. A contract may appear only as part of the *attribute-specifier-seq* of *parameters-and-qualifiers* in a *declarator* (8). The *balanced-token-seq* shall satisfy the syntactic and semantics constraints of an *expression* (5) and shall be contextually convertible to the type bool. Names contained in the *balanced-token-seq* are looked up (3.4) in the context of the declarator. The expression in a contract is potentially evaluated, and entities referenced in the contract are subject to the usual One Definition Rule (3.2). That expression shall be free of side effects, no diagnostic required.
>
> The evaluation semantics of contracts are further expanded in 6.6, 8.4, 12.1. *[Note: contract evaluations may be freely omitted for correct programs with correct data without change in the observable behavior of the abstract machine. –end note]*

Augment paragraph 6.6.3/3 as follows:

> The evaluation of a post-condition contract (if any) is sequenced after the destruction of the local variables. The behavior of the program is unspecified if the post-condition evaluates to false. *[Note: implementations are encouraged to document which behavior they choose, e.g. abrupt termination or continuation with unpredictable behavior. –end note].* Any parameter referenced in the condition is evaluated to the value it has at the point of the post-condition assertion.

Add a new paragraph 8.4.1/9: as follows:

> If a function declaration has a pre-condition contract, the corresponding contract assertion is conceptually part of the function body, and is considered the first statement of the *function-body*. Any parameter in the pre-condition is evaluated to the value of its corresponding argument.

Add a new paragraph 12.1/13 as follows:

> The contract assertion of a pre-condition (if any) of a constructor is executed before its *ctor-initializer*.

Add a new paragraph 3.2/7 as follows:

> If a function declaration in a program has a contract, then its definition shall repeat the same contract. No diagnostic is required if the definition is not in the same translation unit as the non-defining declaration with contract.

7 ACKNOWLEDGMENT

We are grateful to folks who provided feedback on early drafts of this proposal, helping us to make it a stronger proposal. Special thanks to Jonathan Caves, Pavel Curtis, Joe Duffy, Chris Hawblitzel, Aaron Lahman, Neil MacIntosh, Andrew Pardoe, Bjarne Stroustrup, Herb Sutter.

8 REFERENCES

[1] D. Abrahams, L. Crowl, T. Ottosen and J. Widman, "Proposal to Add Contract Programming to C++ (Revision 2)," ISO/IEC JTC1/SC22/WG21, 2005.

[2] L. Crowl and T. Ottosen, "Proposal to Add Contract Programming to C++ (Revision 4)," ISO/IEC JTC1/SC22/WG21, 2006.

[3] J. Lakos, A. Zakharov and A. Beels, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)," 2014.

[4] A. Krzemieński, "Value Constraints," ISO/IEC JTC1/SC22/WG21, 2014.

[5] A. Meredith, "Library Preconditions are a Language Feature," ISO/IEC JTC1/SC22/WG21, 2014.

[6] J. Lakos, A. Zakharov, A. Beels and N. Myers, "Language Support for Runtime Contract Validation (Revision 8)," ISO/IEC JTC1/SC22/WG21, 2014.

[7] G. Dos Reis, S. Lahiri, F. Logozzo, T. Ball and J. Parsons, "Contracts for C++: What Are the Choices?," ISO/IEC JTC1/SC22/WG21, 2014.

[8] Clang. [Online]. Available: http://clang-analyzer.llvm.org/annotations.html.

[9] Microsoft, "Microsoft Source Annotation Language," [Online]. Available: https://msdn.microsoft.com/en-us/library/ms182032.aspx.

[10] [Online]. Available: https://doclazy.wordpress.com/2011/07/14/coverity-suppressing-false-positives-with-cod/][http://stackoverflow.com/questions/3557639/silencing-false-positives-in-coverity-prevent.

[11] J. Lakos, N. Myers, A. Zakharov and A. Beels, "Language Support for Contract Assertions (Revision 10)," ISO/IEC JTC1/SC22/WG21, 2014.

[12] J. D. Garcia, "C++ Language Support for Contract Programming," ISO/IEC JTC1/SC22/WG21, 2014.