p1744r0: Avoiding Misuse of Contract-Checking

By **Rostislav Khlebnikov** and **John Lakos**

Revised June 16th, 2019

ABSTRACT

The C++ contract-checking facility (CCF) was designed as a means to achieve multiple goals: (a) improvement of robustness of software that employs functions with narrow contracts by redundantly checking that such functions are not called out-of-contract at run-time; (b) improvement of software robustness and security by providing supplemental information to the static analysis tools; and (c) improvement of code generation by allowing optimizers to make additional assumptions. All these goals share similar understanding of what the contract-checking statements denote: They document and codify the developer's expectations of what must be true for the program to function correctly.

However, as any other powerful tool in programmer's toolbox, contract assertions may be misused in a variety of ways ranging from easily fixable minor issues to their application to inappropriate problems. In this paper, we demonstrate that CCF misuses may be identified based on two important principles: (I) in a *defect-free* program, function contracts should never be violated, and (II) whether any given aspect of a contract is checked should make no observable (meaningful) difference except, perhaps, runtime performance. We show why heeding these principles is important when employing a CCF for development of production code, how they apply to the intended usage scenarios, and analyze several real-world cases where using a CCF might seem appropriate, but in reality is a mistake. Along the way, we also pay special attention to the cases where strict adherence to the principles might suggest misuse, but which make sense from a practical engineering standpoint, and can therefore be deemed acceptable.

INTRODUCTION

When defining an interface of a library function, the developer is faced with a decision of whether to impose limitations on semantically valid inputs that have to be respected before the function invocation. In many cases, the most reasonable decision is to identify and clearly document the preconditions, leave the behavior undefined should the preconditions be unmet, and implement *defensive checks* to allow early detection of program defects leading to out-of-contract calls.[1]

Breach of a contract, i.e., a caller failing to satisfy all preconditions of a function or failures of the function itself to satisfy all of its postconditions when all its preconditions are satisfied, is necessarily a defect in the program – on part of the function caller or its author respectively. As such, function contracts should *never* be violated in a ***defect-free*** program, regardless of any external input that the program receives. In reality, however, code often contains bugs leading to inadvertent out-of-contract function invocations. Contract *checks* are *extra* code that allow detecting such violations early in the development process, facilitating their eradication and, ultimately, helping to improve the robustness of software.

Since (defensive) contract checks are redundant, it should be possible to disable them, e.g., to attain better performance, once the program owner has sufficient

[1] See **p1743: *Contracts, Undefined Behavior, and Defensive Programming*** paper for an in-depth analysis of the rationale behind such decisions.

confidence that no contracts are violated. To maintain this ability, however, it is also essential to ensure that the predicates of the contract checks themselves have no side effects[2] that affect the essential behavior of the program. Furthermore, lack of side effects in the predicates combined with understanding that contract assertions should never be violated in a correct program, significantly simplify reasoning about the code containing the defensive checks: For the code to function properly, all the predicates must evaluate to `true` and no additional control flow paths are introduced by the contract checks. Depending on the goals of the reader, the contract checking statements could be summarily ignored to focus the attention on the essential behavior of the function, or could serve as additional clarification of the code.

Analyzing code in terms of whether these principles hold, allows identifying scenarios where the contract-checking statements can be relied upon and where they are a wrong tool for the job.

INTENDED USE SCENARIOS

Runtime Contract Checking: One of the primary uses for a CCF is detection of out-of-contract invocations of functions with narrow contracts. Consider a function that processes a sorted sequence of integers in a half-open range `[0, 100)`:

```
void processSequence(const std::vector<int>& sequence);
    // Process the specified 'sequence'.  The behavior is undefined unless for
    // every element 'e' of the 'sequence', '0 <= e < 100'.
```

To help detect inadvertent out-of-contract calls, the implementation of `processSequence` might have defensive precondition checks:

```
void processSequence(const std::vector<int>& sequence)
    [[pre: std::all_of(sequence.begin(), sequence.end(), in_range{0, 100})]]
```

The code invoking this function correctly will ensure that the preconditions are satisfied by construction. If a caller of `processSequence` fails to do so, indicating a program defect, a contract violation will be detected by the defensive checks in certain specific build modes, leading to invocation of the violation handler, which will typically fail-fast, immediately alerting the developer of the exact point of failure (simplifying the analysis) early in the development process:

```
std::vector<int> sequence(50);
std::default_random_engine eng;
std::uniform_int_distribution<int> dist{1, 100}; // BUG: should be {1, 99}

std::generate(sequence.begin(), sequence.end(), [&] { return dist(eng); });

processSequence(sequence); // Defect detected close to the source
```

Once the defects are fixed, however, the precondition checks may be removed completely without having *any* effect on the essential behavior of the program except, perhaps, improved performance.

Static Analysis: Another important use for contract annotations is to inform the static analyzers of the pre- and postconditions allowing them to detect code paths

[2] Note that in the current WP any side effects in contract-checking statements are considered UB. Our arguments, however, are based on extensive experience with a macro-based CCF, and would equally apply to C++ language-based CCF should P1670 be adopted.

necessarily leading to out-of-contract calls without a need to run the application. Static analyzers are capable of detecting potential contract violations that might only occur at run time in very specific corner cases, allowing to further improve robustness and security of the software. In addition, static analysis tools are capable of using predicates that cannot be checked at run time (e.g., predicates that cannot be implemented, have infeasible run-time cost, or inevitable significant side effects), supplementing run-time checking even further:

```
class multiset {
    // . . .
    std::pair<iterator, iterator> equal_range(const Key& key)
        [[post axiom r: is_reachable(r.first, r.second)]]; // unimplementable
};

template<class ITERATOR, class FUNC>
void for_each(ITERATOR first, ITERATOR last, FUNC func)
    [[pre axiom: is_reachable(first, last)]];

using Data = multiset<Datum>;
using DataIt = Data::iterator;
Data data;
// . . .
std::pair<DataIt, DataIt> range = data.equal_range(someKey);
for_each(range.first, range.second, &processDatum);  // OK
for_each(range.second, range.first, &processDatum);  // static analyser warning
```

Similar to run-time checking, however, should a program be *defect-free*, even physically removing the contract annotation from source code would not affect the essential behavior of the program.

Improved Code Generation: Contract checking statements (CCSs) are in essence statements of what must be true in the program. If the optimizer is instructed to treat violations of such statements as (language) undefined behavior, it can use them to generate more efficient code. For example, consider a function that performs highly efficient fast Fourier transform, but relies on data being properly aligned for vectorization:

```
void FFT(std::complex<float>* result, std::complex<float>* signal, int nSamples)
    [[pre: 0 == reinterpret_cast<std::uintptr_t>(signal) % 64]]
    [[pre: 0 == reinterpret_cast<std::uintptr_t>(result) % 64]]
    [[pre: 0 == nSamples % (64 / sizeof(std::complex<float>))]];
```

With these preconditions assumed to evaluate to `true`, the compiler may generate minimal vectorized machine code. Having such function requirements codified as precondition checks not only to positively affects code generation, but also allows such assumptions to be easily both checked at run time and taken in consideration during static analysis prior to using them for optimization, without requiring any changes to the source code: Same annotations can be used to improve both robustness *and* performance. And again, should the contract annotations be removed from a *defect-free* program, the only observable change would be a change in run-time performance, but the essential behavior would stay identical.

POTENTIAL PITFALLS IN CORRECT USE SCENARIOS

Reliance on Side Effects: Even if the program is correct and no contracts are violated at run time, contract predicates having side effects that impact the essential program behavior not only complicates reasoning about the code by

harboring non-redundant side effects and possibly additional flow control (e.g., throwing an exception), but also takes away the program owner's ability to create high-performance builds with contract checks disabled:

```
[[assert: setOfIntegers.insert(value).second]];
```

Such code would have no effect if the contracts are not checked at run time, altering the essential program behavior. In some cases, the side effects might be more subtle, but still have a significant impact:

```
int encrypt(int value);
int decrypt(int value);
int corrupt(int value);
bool isCorrupted(int value);

class EncryptedStore {
    std::map<int, int> d_map;

  public:
    int getValue(int index) const { return decrypt(d_map.at(index)); }

    void setValue(int index, int value) { d_map[index] = encrypt(value); }

    void corruptValue(int index)
        [[pre: !isCorrupted(d_map[index])]] // Has a side effect!
    {
        d_map[index] = corrupt(getValue(index));
    }
};
```

When the precondition check of `corruptValue` is run, an element is inserted into the map if it is not already present. This leads to very different behavior of `getValue`, that `corruptValue` invokes, in case the `index` is not present in the map – if the contract checks are enabled, `getValue` will simply observe `0`, whereas if contract checks are disabled, `getValue` will throw a `std::out_of_range` exception.

In some cases, however, it is acceptable to allow the predicates to have observable side effects that do not alter the essential function behavior. For example, logging the object state or performing a check that might require a temporary heap allocation, might be acceptable, especially when such side effects are temporarily introduced during the routine process of code development and maintenance:

```
class HttpHeaderFields {
    std::vector<Field> d_fields;

  public:
    bool contains(std::string_view name) const
    {
        LOG_TRACE << "Checking whether '" << name << "' is present among "
                  << d_fields.size() << " fields.";
        // . . .
    }

    void addField(std::string_view name, std::string_view value)
        [[pre: !contains(name)]]  // Writes to log
        [[pre: "content-length" != name ||
               0 <= std::stoi(std::string(value))]]  // Potentially allocates
    {
        // . . .
    }
};
```

While technically with such side effects, the observable behavior of the program would differ in different build modes (different log output or slightly changed memory layout), it does not violate the spirit of the principle in that the core behavior (the primary purpose) is not affected by the presence of contract checks.

SCENARIOS WHERE CCF IS UNSUITABLE

Input Validation: Contract violations always indicate a programming error, should be reported to the developer, and, in general, cannot be recovered from by the running application (since the programmer's expectations were not met). Contract assertions are, therefore, a poor choice for checking the validity of any external input to the program – e.g., command line arguments, data read from configuration files, or received over-the-wire:

```
int main(int argc, const char* argv[])
{
    [[assert: 1 < argc]];

    std::ifstream dataFile(argv[1]);
    [[assert: dataFile]];

    int rowCount, columnCount;
    dataFile >> rowCount >> columnCount;
    [[assert: dataFile && 0 < rowCount && 0 < columnCount]];

    // . . .
}
```

Since the program has no control of such data, it is impossible for the developer to avoid violations of such contracts, and disabling the contract checks would more often than not lead to hard UB. Such issues might arise not only when using the CCSs to test the data validity directly, but also when passing unsanitized external input to functions having narrow contracts. Consider the following scenario of use of `processSequence`:

```
std::vector<int> sequence;
std::copy(std::istream_iterator<int>(std::cin),
          std::istream_iterator<int>(),
          std::back_inserter(sequence));

processSequence(sequence);
```

Attempting to circumvent the core problem by installing a custom violation handler does not resolve the issue of contracts being violated during normal execution:

```
std::vector<int> sequence;
std::copy(std::istream_iterator<int>(std::cin),
          std::istream_iterator<int>(),
          std::back_inserter(sequence));

set_violation_handler([](auto&&) { throw ContractViolationException{}; });
try {
    processSequence(sequence);
}
catch (const ContractViolationException& e) {
    std::cout << "Received an incorrect sequence.";
}
```

While this code might work as expected in some build modes, its behavior will radically change should the application be built with all or some of the contract

checks disabled.

Note that in some cases, separating input validation from function execution might be impractical. For example, consider a function that builds an abstract syntax tree of C++ code represented as a string:

```
AST build_AST(std::string_view sourceCode)
    [[pre: is_valid_cpp_code(sourceCode)]];
```

Any client of `build_AST` would be required to ensure that data, typically read from a file, is valid C++ code before passing it to `build_AST`, which would essentially double the time required to attain the result. This indicates that such a function would be better served to have a wide, rather than a narrow, contract:

```
std::expected<AST, ParseError> build_AST(std::string_view sourceCode);
    // Return an 'AST' on success and a 'ParseError' otherwise.
```

Replacement for English Contracts: Since a language-based CCF that allows the predicates for pre- and postconditions to be attached to the function declaration might suggest that they can replace the natural-language function contracts in their entirety, it is important to highlight several significant limitations in the expressivity of CCS. Descriptive names of the function and its arguments in combination with pre- and postconditions might be good enough for relatively simple functions:

```
double squareRoot(double value)
    [[pre: 0.0 <= value]];
```

For more complex functions, however, many intricacies of the full function contract might become difficult to communicate using CCSs. Consider a contract that a `logMessage` function of the `Logger` class might have:

```
class Logger {
    // . . .
    void logMessage(const Category& cat, Severity svr, Record *record);
        // Log the specified '*record' after setting its category attribute to
        // the name of the specified 'cat' and severity attribute to the
        // specified 'svr'.  (See the component-level documentation of
        // 'ball_record' for more information on the fields that are logged.)
        // Store the record in the buffer held by this logger if 'svr' is at
        // least as severe as the current "Record" threshold level of 'cat'.
        // Pass the record directly to the observer held by this logger if
        // 'svr' is at least as severe as the current "Pass" threshold level of
        // 'cat'.  Publish the entire contents of the buffer of this logger if
        // 'svr' is at least as severe as the current "Trigger" threshold level
        // of 'cat'.  Publish the entire contents of all buffers of all active
        // loggers if 'svr' is at least as severe as the current "Trigger-All"
        // threshold level of 'cat' (i.e., via the callback supplied at
        // construction).  The behavior is undefined unless 'record' was
        // previously obtained by a call to 'getRecord' on this logger.  Note
        // that this method will have no effect if 'svr' is less severe than
        // all of the threshold levels of 'cat'.  Also note that 'record' must
        // not be reused after invoking this method.
};
```

Many aspects of such a contract, and especially the details of the essential behavior of the function, would often require introduction of additional predicates that, if implemented, would necessitate storage of additional state, or would need to be left unimplemented and only be used in `axiom`-level CCSs only:

```
void logMessage(const Category& cat, Severity svr, Record *record);
  [[pre  axiom: is_valid_record_ptr(this, record)]]
  [[post axiom: svr < Severity::Record     || is_buffered(d_buf, record)]]
  [[post axiom: svr < Severity::Pass       || is_published(d_observer, record)]]
  [[post axiom: svr < Severity::Trigger    || is_published(d_observer, d_buf)]]
  [[post axiom: svr < Severity::TriggerAll || is_published_all(d_publishAllCb)]]
  [[post axiom: !is_valid_record_ptr(this, record)]]
```

Furthermore, for most non-trivial member functions, the contract predicates would need to access encapsulated implementation details of the class, not only unnecessarily exposing them to the client, but also requiring their thorough understanding for correct use. Such exposition increases the likelihood that the clients will depend on these details, making it much more difficult to add, remove, or modify the implementation during maintenance.

Finally, checks even for some simple contract clauses cannot be implemented in a straightforward manner with the CCF specified in the current WP, because specification of necessary functionality has been deferred. For example, documenting that `vector<T>::push_back()` increases its `size` by exactly `1`, cannot currently be replicated using a CCS without workarounds.

Substitute for Testing: While contract checks might supplement unit and integration tests by helping to uncover bugs in function implementations and in the test drivers themselves (e.g., postcondition and assertion violations indicating flaws in internal logic, and precondition violations indicating incorrect use of other functions), attempting to replace testing with even the most thorough set of CCSs is ill-conceived. Corner cases might not be checked until the system arrives into a very rare specific state (possibly in production, potentially leading to disastrous results especially if the code is built with contract check disabled), reproducing the failure state might be inherently difficult, repeating the tests after code changes becomes an issue again, pinpointing the root of the problem might be unnecessarily hard because essentially the entire application is the code “under test”, etc. In essence, while an untested program that has contract checks in place might be marginally easier to maintain than an untested program without them, it would still suffer from the same well-known drawbacks that rigorous testing aims to alleviate.

CONCLUSION

In order to attain all the benefits afforded by a contract-checking facility, it is crucial to identify the scenarios for which it is the right tool and those where using a CCF would be counterproductive or even dangerous. Any correct application of a CCF should abide by the two principles: (I) in a *defect-free* program, function contracts should *never* be violated, and (II) its essential behavior should remain unchanged whether or not contracts are enabled at all. Heeding these principles prevents introduction of hidden control flow paths and side effects, thereby significantly simplifying reasoning about both the functions that contain contract-checking statements as well as their callers, while enabling production of builds with all checks disabled (or even used for optimization), squeezing every last bit of performance possible.