Contracts, Undefined Behavior, and Defensive Programming

BY **ROSTISLAV KHLEBNIKOV** AND **JOHN LAKOS**

Revised Thursday, February 1, 2018

ABSTRACT

When designing libraries, especially (hierarchically) reusable ones, developers are often faced with a decision regarding the domain of semantically valid input that a function they create will accept. Consider designing the C standard library function `strlen`:

```
1 size_t strlen(const char *str)
2 {
3     if (!str) return 0;    ⟵ Is this a good idea?
4     size_t count = 0;
5     while (str[count])
6         ++count;
7     return count;
8 }
```

In this paper, we aim to convince readers that such a test for a null pointer in `strlen` is misguided and, more generally, that creating functions having *artificially* defined behavior for every syntactically legal combination of inputs has negative implications for performance, correctness, and stability. Instead of widening the input domain, we recommend (1) identifying preconditions, i.e., those conditions that a caller is required to satisfy whenever invoking the function, (2) leaving the behavior undefined whenever those preconditions are not met, and (3) employing an assertion facility to detect (in appropriate build modes) program defects that result in function misuse. Following these guidelines facilitates the development of robust high-performing reusable libraries.

INTRODUCTION

When implementing software, developers encounter two disjoint categories of functions: those that can naturally accept any syntactically legal inputs and can be called in any valid state (program and/or object state, if it is a member function), and those that have some semantic limitation on either (or both). For example, `std::vector<T>::size` can be called on a vector in any state, whereas `std::vector<T>::front` requires a non-empty vector.

There are two common approaches to implementing a function that has semantic limitations on its syntactically legal inputs or state. The first is to check whether semantic requirements for the function call are satisfied, and "handle" (i.e., report) any obviously incorrect inputs that are detected, e.g., by throwing an exception or returning an error code. The second is to rely on the caller to invoke the function appropriately, supplying only proper inputs given the current state of the program. In other words, the implementer has to decide whether to always explicitly check if the input is proper and somehow deal with it if it is not, or just impose ***preconditions***, i.e., what is required of a function's arguments and the state of the program (including object state) for a function call to be valid.

The set of preconditions and ***postconditions***, i.e., what must happen as a function of state and arguments, along with ***essential behavior*** a function guarantees, form a ***contract*** between the function and its clients. A function having no preconditions is said to have a ***wide contract***; otherwise, it has a ***narrow contract***. Narrow contracts imply that the behavior of the function in cases where its preconditions are violated is, at least as far as the immediate client knows, ***undefined***.

Drawbacks of Artificially Wide Contracts

Some functions, such as `std::vector<T>::push_back` or `std::vector<T>::clear`, have naturally wide contracts, and it is ideal to allow such functions to have a wide contract. A large proportion of functions we encounter in practice, however, have some form of precondition, e.g., `std::vector<T>::pop_back`, which requires a non-empty vector. It may seem that narrow contracts are inherently dangerous, and that widening the contracts of such functions to avoid the accidental corruption of the program state should be the "go to" policy for high-quality library design. For example, a function computing the square root of a given number may throw an exception if its argument is negative:

```
1 double squareRoot(double value)
2 {
3     if (value < 0)
4         throw std::invalid_argument("Negative value passed to squareRoot.");
5     // Compute the square root
6 }
```

Such an implementation of `sqareRoot` artificially widens what is naturally a narrow contract. There are several serious drawbacks to making inherently narrow contracts *artificially* wide.

Feasibility: In some cases, checking the preconditions cannot be done in any reasonable amount of time. For example, testing whether a (stateless) comparator passed to a sorting routine provides a strict weak ordering on 64-bit integers would take years. There are even some preconditions, such as that a given character string be null-terminated, that generally cannot be checked at all.

Efficiency: Checking the preconditions may be computationally expensive compared to the productive work being done. For example, testing whether the input range is sorted before performing a binary search will increase the run-time complexity from `O[log(N)]` to `O[N]`, which might even violate the essential behavior promised by the contract. Even when the checks do not change the big-O complexity of the original algorithm, they still impose a performance penalty, which – unlike for the theoretical algorithmic analysis – matters for industrial software. Furthermore, this performance penalty is present on every function call, even for the cases where the caller is certain that the state and the arguments passed to the function satisfy all of its preconditions. Finally, these checks cannot be elided under any circumstances because they are part of the defined behavior of the function, which can be relied upon by client code; removing these checks in, say, an optimized build mode, would necessarily change the essential behavior of the function and potentially the semantics of the program.

Reliability: Artificially wide contracts tend to mask program defects. Trying to eliminate preconditions by 'fixing' the improper input (e.g., `strlen(nullptr)` returning 0), returning an error code, or silently doing nothing (e.g., `std::vector<T>::pop_back` returning normally if the vector is empty) is ill-conceived: instead of allowing us to detect such errors automatically in certain build modes, it forces the error-masking behavior to be present in every build mode. The client's error is then likely to be propagated through the function's call chain, possibly for quite some time, before (if

we're lucky) it manifests, at which point the source of the error may be difficult to uncover.

Maintainability: Always having to check for, and then somehow handle, nonsense inputs in production code not only increases the amount of code to be designed, compiled, documented, tested, and executed, but also needlessly hinders code readability and maintainability. The problem is further aggravated by the necessity to support and test various combinations of error conditions.

From the standpoint of an application developer, reasoning about and maintaining application code becomes even more complicated when multiple libraries take *different* approaches to handling semantically incorrect inputs.

Extensibility: If the behavior of a function is delineated for obviously incorrect inputs – e.g., the documentation states that the function throws if the input is outside of the "required" domain – that behavior becomes part of the essential behavior for that function, which makes it far more difficult to improve the code to accommodate a wider set of useful inputs. Providing new, useful functionality would now involve changing the contract and breaking backwards compatibility with existing clients, rather than merely widening (i.e., adding to) it.

Overall, adopting narrow contracts for functions that naturally have preconditions helps us to develop high-performance, robust, maintainable, and highly reusable library code.

Narrow but not Too Narrow Contracts

Just as artificially wide contracts have their drawbacks, so do artificially narrow ones. That is, if the natural implementation of a function would support an edge case, it is both more efficient and more maintainable to subsume the edge case in the function's defined behavior and let it be handled there. For example, consider a member function of a sequence container whose purpose is to replace a subsequence of elements with a specific value:

```
1 void replace(int index, const TYPE& value, int numElements);
```

While it may seem natural to impose a precondition of `'1 <= numElements'` (i.e., at least one element should be replaced), a natural implementation of the function can handle replacing zero elements without any changes. Relaxing the constraint to accept non-negative `numElements`[1] might, however, be beneficial for clients by allowing them to avoid unnecessary checks.

"Soft" versus "Hard" Undefined Behavior (UB)

In any reasonably large code base, it is inevitable that a function will eventually be called without meeting its preconditions due to a program defect. In practice, a function called ***out of contract*** (i.e., invoked with one or more of its preconditions violated) immediately results in what is called ***library undefined behavior (soft UB)***, which – left unchecked – may subsequently trigger ***language undefined behavior (hard UB)*** – the latter being what most people mean when they say the phrase "undefined

[1] The set of preconditions would then become `'0 <= numElements && 0 <= index && index + numElements <= length()'`, where `length()` returns the number of elements in the sequence.

behavior".

Hard (language) UB results from attempting to perform a native C++ operation in a manner that the C++ Language specification designates as *undefined behavior*. *Undefined* behavior is qualitatively more severe than just *unspecified* or *implementation-defined* behavior. Although invoking hard UB is never good, the existence of hard UB in C++ allows compilers to generate code having higher performance than would be possible if all behavior in the language were somehow defined. Unlike for soft (library) UB, if the compiler can recognize conditions under which hard UB would occur, it is free to assume that these conditions will never be encountered in practice, allowing it to generate more compact, faster machine code. In fact, if the compiler can deduce a code path that will necessarily lead to hard UB, it can – as an optimization – summarily elide all that code from the translation unit with no diagnostics required!

From the perspective of programming using functions that have narrow contracts, however, hard UB is *not* triggered immediately on entry to the function called out of contract. Consider the following simple implementation of `strlen`, having a *properly narrow* contract:

```
1  size_t strlen(const char *str)
2      // Return the length of the specified 'str'. The behavior is
3      // undefined unless 'str' is null-terminated.
4  {
5      ⟵ Library (but not language) UB might have already occurred here.
6      size_t count = 0;
7      while (str[count])    ⟵ Language UB might be triggered here.
8          ++count;
9      return count;
10 }
```

While hard UB will be triggered immediately at line 7 if `str` is null (or eventually if `str` is not null-terminated), the program state will remain valid until control flow reaches that line, which enables us on entry into the function body (in certain specific build modes) to detect and report – *but neither handle nor hide* – client misuse of the function. This robust runtime technique for automatically detecting misuse by our clients is generally known as ***Defensive Programming***.

Defensive Programming

Defensive Programming amounts to providing (in certain specific build modes) *extra* code to help ensure that the *trusted* programmatic clients (within the same process) do not invoke a function and inadvertently fail to satisfy one or more of its preconditions. The runtime checks performed by this code can be activated to help identify latent defects early in the development process and also after the software has been released. These checks (along with any action taken should a check fail) are, however, *never* a part of a function's contract, cannot be relied upon by the clients, and may be added, removed, or changed at any time without notice. Furthermore, for a ***defect-free*** program, they have no side effects other than increased runtime (same as that of artificially wide contracts), and, apart from the checks themselves, do not directly affect the instruction stream of a single thread of execution. Therefore, these checks are

redundant and can be turned off without *any* change in the semantics of the program.

The classical approach to implementing defensive checks is to use a C-style `assert`, which outputs diagnostic information to `stderr` and calls `abort`, but does so only when `NDEBUG` is not defined. The preconditions for the function call that are both checkable and appropriate to be checked should be asserted at the beginning of the function body:

```
1 void Date::setYearMonthDay(int year, int month, int day)
2 {
3     assert(isValidYearMonthDay(year, month, day));  ⟵ (Redundant) check.
4
5     // Implementation follows
6 }
```

Remember that not all preconditions can be checked, not all of those that can be checked are feasible, and not all those that are feasible are worthwhile:

- The function cannot check for something where it does not have sufficient historical information to determine if the (possibly implicit) precondition has been met. For example, there is generally no way to detect that a supplied object has not already been destroyed.
- Even if the precondition can be checked, the cost may be prohibitive in any build mode. For example, determining whether a binary relation on two 64-bit integers is transitive.
- Even if the cost of checking is not outrageous, it may nonetheless exceed the big-O complexity of the useful work being done by the function and, thereby, perhaps even violate essential behavior promised by the function's contract. For example (as before), determining whether the input range for a binary search function is sorted. (Note that having a separate "auditing" assertion level to address this specific scenario would be valuable.)
- Even if the big-O complexity of the check is not greater than that of the useful work being done by the function, and therefore entirely feasible to implement, the implementer may wisely choose to omit the (typically implicit) precondition check on the basis that it would not solve any problem likely to occur at runtime. For example, it could be easily checked whether an object passed in by pointer or reference is sufficiently aligned, but that should never be necessary as the C++ language's (static) type system obviates that as a problem to be addressed at runtime.

All remaining precondition checks – i.e., those that are doable, tractable, not disproportionately expensive, and potentially useful – should be codified by asserting the preconditions in the appropriate build modes.

Even such a simplistic assertion mechanism allows detecting out-of-contract function calls at the earliest possible point, thus greatly simplifying the debugging process. Unfortunately, C-style assertions are not sufficiently configurable with respect to (1) assertion-level build modes, and (2) actions to be taken if an assertion fails. BDE provides a better, much more flexible alternative in the form of `BSLS_ASSERT*` macros, which are described in detail in a separate paper: ***Defensive Programming using***

BSLS_ASSERT. See also the component documentation in `bsls_assert.h`[2] and the proposal for C++ standardization ***Support for contract based programming in C++***[3].

Input Validation versus Defensive Programming

Input validation is sometimes incorrectly conflated with Defensive Programming. Input validation is concerned with testing untrusted (typically out-of-process) input – e.g., from a command line, configuration file, or network connection – to ascertain whether or not that input meets certain criteria. It is part of the essential behavior of a function, is not redundant, and must ***always* execute** in ***all* build modes**. In contrast, defensive programming is concerned with *redundantly* checking that clients invoke functions in contract with the goal of exposing program defects. External input should *never* be validated using an assertion mechanism.

For example, consider a function that parses a configuration file:

```
1 int readConfigFile(Config *result, const char *filename);
```

The function itself may have a narrow contract (i.e., it may require that the `result` points to a valid `Config` object, and `filename` points to a null-terminated string) and should therefore assert these specific preconditions to detect misuse by programmatic clients. The *contents* of the indicated file, however, are out-of-process inputs and such inputs should (almost) ***never*** be propagated into the trusted region of the running process; instead, they should be ***thoroughly*** validated and any errors should be reported to the caller – e.g., via a return code or by throwing an exception.

Validating and Non-Validating APIs

From the standpoint of designing reusable-library APIs, it is often beneficial to have two versions of a function: one having a narrow contract for clients that know their input is valid, and one having a wide contract for clients that call it with untrusted data. Furthermore, it is often necessary to also provide a means for client code to check complex preconditions directly. For example, consider a subset of the API for a typical `Date` class that deals with setting the date given its respective year, month, and day values[4]:

```
1  class Date {
2      // ...
3    public:
4      static bool isValidYearMonthDay(int year, int month, int day);
5          // Return 'true' if the specified 'year', 'month', and 'day'
6          // collectively represent a valid value for a 'Date' object,
7          // and 'false' otherwise
8      void setYearMonthDay(int year, int month, int date);
9          // ... The behavior is undefined unless 'year', 'month', and
10         // 'day' collectively represent a valid 'Date' value. ...
11     int setYearMonthDayIfValid(int year, int month, int day);
12         // Set this object to have the value collectively represented by the
13         // specified 'year', 'month', and 'day' attributes if they
```

[2] https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_assert.h
[3] http://wg21.link/P0542
[4] For more information on proper contract documentation see the separate paper: ***Delineating C++ Contracts in English***.

```
14          // correspond to a valid 'Date' value (see 'isValidYearMonthDay').
15          // Return 0 on success, and a non-zero value (with no effect) otherwise.
16  };
```

Given the methods above, we can then proceed to implement a function that, say, reads a date from a file in two valid ways:

```
1  int readDateFromFile(Date *result, const char *filename)
2  {
3      int y = -1, m = -1, d = -1;
4      ifstream(filename) >> y >> m >> d;
5      if (!Date::isValidYearMonthDay(y, m, d)) {
6          return -1;  // failure
7      }
8      result->setYearMonthDay(y, m, d);
9      return 0;  // success
10 }
```

or in a more compact, less error prone, and potentially more efficient way:

```
1  int readDateFromFile(Date *result, const char *filename)
2  {
3      int y = -1, m = -1, d = -1;
4      ifstream(filename) >> y >> m >> d;
5      return result->setYearMonthDayIfValid(y, m, d);
6  }
```

What must ***never*** happen is the assertion that the date read from the file is valid:

```
1  int readDateFromFile(Date *result, const char *filename)
2  {
3      int y = -1, m = -1, d = -1;
4      ifstream(filename) >> y >> m >> d;
5      assert(Date::isValidYearMonthDay(y, m, d));    ⟵ Very bad idea!
6      d->setYearMonthDay(y, m, d);
7      return 0;  // Date loaded successfully.
8  }
```

With assertions disabled, the function above could easily return an invalid date while reporting it as valid, leading to unbounded liability for its clients. For a much more in-depth treatment regarding the critical distinctions between *Input Validation* and *Defensive Programing*, see the separate paper entitled ***When, and When NOT, to use Assertions***.

CONCLUSION

When implementing a reusable-library function having natural preconditions, *narrow* contracts are to be preferred over *artificially wide* ones. An appropriately narrow contract reduces costs associated with development and testing, improves performance, reduces object-code size, and allows useful behavior to be added as needed. A narrow contract implies the existence of *undefined behavior*, which, if inadvertently invoked, can be difficult to diagnose. Narrow contracts in combination with *Defensive Programming*, however, enable detecting and reporting many programming errors early in the development process without necessarily (or permanently) impacting the performance of the final product in production. Overall, narrow contracts – used *properly* – allow us to build faster, easier-to-understand, and ultimately *more reliable* hierarchically reusable libraries than would otherwise be achievable.