P2053R1
2020-08-14

Rostislav Khlebnikov: rkhlebnikov@bloomberg.net
John Lakos: jlakos@bloomberg.net

Defensive Checking Versus Input Validation

ABSTRACT

For a software system to function as intended, the assumptions its implementers make in the source code must be satisfied. Such assumptions fall into one of two distinct categories: (1) those that are always expected to be true in a properly implemented system, and (2) those whose truth depends on factors external to the system. Violations in each category indicate fundamentally different issues: software defects for the first category and malformed input for the second category. Thus, a well-designed software system should employ different mechanisms for each category. Empirical evidence suggests, however, that practitioners often confuse and/or conflate these categories and use an incorrect mechanism, which leads to both reduced efficiency and potentially catastrophic failures.

This paper first illustrates the important differences between the *defensive* and *input validation* runtime checks that are intended to verify the truth of assumptions in the two categories. The detrimental consequences of conflating the two checking mechanisms are then shown. Finally, criteria are provided to discriminate between the two categories based on physical *neighborhoods* as well as the packaging, testing, deployment, and consumption (PTDC) requirements that such neighborhoods must satisfy. Only those checks that (1) are *defensive in nature* (i.e., that verify the assumptions that are always expected to be true) and (2) are *manifestly defensive* (i.e., that can be statically proven) when embedded in their physical neighborhoods can be implemented using defensive-checking frameworks such as `<cassert>`.

INTRODUCTION

The vast majority of software systems accept, in various forms, certain structured data as input. The software systems then process that data in accordance with their respective specifications, often employing external resources in the process. Result data is produced as their output. This data-transformation process may be disrupted in three disparate ways:

(1) the software system itself might contain defects, thereby preventing proper data processing,
(2) the input data might be malformed, i.e., syntactically or semantically inconsistent with the system's specifications, and
(3) the external resources required for processing the data might not be available.

From the developer's perspective, these disruptions belong to two fundamentally different categories.

- Software defects are under the developer's control. They can be detected and remediated, and, with proper engineering discipline, the defect rate can reasonably be expected to decrease over time.

- Malformed input data and the lack of external resources are beyond the developer's control. The likelihood of the system encountering these issues remains largely consistent over time. In today's security-aware world, the need to validate external inputs never abates.

A well-designed software system will often contain runtime mechanisms for both verifying its own correctness and validating its external inputs. The correctness checks defend against the unintended yet inevitable defects introduced during software development and subsequent maintenance. Such *defensive checks* are entirely redundant to the functional specification of a software system, i.e., they have no effect on its *essential behavior.*[1] Hence, defensive checks can potentially be removed (e.g., by using specific build modes) once the program owner is sufficiently confident that the software is free of defects. In contrast, input validation is concerned with ensuring that data entering from outside the trusted region of a subsystem satisfies the requirements imposed by that subsystem. Throughout this paper, both the data supplied by the software system's users and the external resources accessed by the software system itself will be collectively referred to as *input*. Such improper input is likely to be encountered in production regardless of how hardened the software system itself becomes. Therefore, input-validation checks are a necessary part of the program's essential behavior and can *never* be removed from the final product.

Due to consequential differences between defensive checks and input validation, conflating them could easily render the software unfit for its purpose. Using a defensive-checking framework — such as `<cassert>` or Python's `assert` statements — to implement input validation might allow malformed input to harm the system when it is built with defensive checks disabled. Such misuse could lead to incorrect output, hard-to-diagnose crashes, and perhaps even vulnerability to malicious attacks. Furthermore, if this mistake is pervasive in a code base, it might lead to the inability of application owners to disable *any* defensive checks for fear that needed input-validation checks will be disabled as well.

A production program that always attempts to validate (and, if defective, circumnavigate) the internal logic of a program at run time (e.g., by using hard-coded, unconditional checks) can also be costly and problematic.

- Attempting to recover from a program defect (and to continue normally after one has been detected) is a dubious engineering practice. Even if code is carefully and specifically crafted to detect and handle a defect, the program would have to fail in an unforeseen way for the detection and recovery to work as intended, implying that the program simply cannot reliably uncover the true source and

[1] The *essential behavior* of a software system is the behavior that is mandated by that system's specification, sometimes referred to as its *contract*; see **khlebnikov19a**.

impact of a defect at run time.[2]

- Additional code for handling a defect is not run unless the program is defective. Therefore, such code is difficult (if not intractable) to test, is typically never executed, and is especially likely to contain latent defects itself.
- These defensive checks are implemented with no consideration of whether they might become redundant. Hence, they will be present even in a defect-free program, leading to a needless performance penalty in perpetuity.
- Finally, defect handling, i.e., attempting to *recover* from defects as opposed to merely *detecting and reporting* them, affects not only functions that perform the checks, but also *all* of their callers. The resulting combinatorial explosion of failure modes and greatly increased code complexity generally leads to an undesirable maintenance burden. The desire to navigate around all possible failure modes of even a simple function results in bloat, which Figure 1 illustrates.

```
float min_float(const float *first, const float *last)
    // Return the minimal value in the specified range '[first .. last)'.
{
    if (nullptr == first)           throw std::invalid_argument{"null first"};
    if (nullptr == last)            throw std::invalid_argument{"null last"};
    if (std::less<>{}(last, first)) throw std::invalid_argument{"bad range"};
    if (first == last)              throw std::invalid_argument{"empty range"};

    const float *cur = first;
    const float *min = first;
    while (++cur < last) {
        if (std::isnan(*cur))      throw std::domain_error{"unexpected NaN"};

        min = *min < *cur ? min : cur;
    }

    if (min < first || last <= min) throw std::logic_error{"algorithm failed"};
    if (cur != last)                throw std::logic_error{"algorithm failed"};
    return *min;
}
```

Figure 1: Attempting to handle (rather than merely detect) software defects introduces significant additional complexity.

The superficial similarities between the two kinds of checks can lead developers treat them interchangeably despite their important differences. This is more than merely an academic concern: Empirical data from extensive use within Bloomberg[3] confirm

[2] Note the practical distinction between a defensive check in newly minted software and a new defensive check in battle-hardened software: Proceeding after a defensive-check failure in unproven new code would be foolish, yet very practical reasons exist for proceeding (reporting and continuing) after the failure of a new check in otherwise known-to-be-working code.

[3] Bloomberg's internal defect reports indicate that misuse of defensive checking to perform input validation is a frequent reason for major outages. The authors have no quantifiable data from outside Bloomberg but have no reason to believe that such data would be contradictory.

that these misjudgments occur all too frequently in practice with sometimes catastrophic results. The following section provides criteria that a developer can use to determine whether checking a particular assumption can be properly classified as redundant and therefore optional. Employing a defensive-checking framework only for checks that satisfy these criteria and never for checks that are essential and therefore always mandatory is key to creating robust, maintainable, and high-performance software.

Discriminating Between Defensive Checks and Input Validation

The primary criterion for classifying a check as defensive is that the assumption that the check verifies is expected to always be true during operation of the software system.[4] If this criterion is not met, checking the assumption is essential. For example, a typically unreasonable assumption would be to expect that external input supplied by a human operator (let alone an unknown remote client making requests over the Internet) will satisfy any limitations required for an assumption to be true. Checks that validate such assumptions are therefore an essential part of the software system. Implementing such checks using a defensive-checking framework would be misguided. In contrast, some assumptions, e.g., loop invariants, are always expected to be true. Whether and to what extent to perform any redundant runtime verification of assumptions of this kind is at the developer's discretion. If the developer chooses to introduce such checks, referred to as *defensive in nature*, the checks can reasonably be implemented using a defensive-checking framework (e.g., `<cassert>`).

In a properly implemented system, every assumption that is expected to always be true should be — at least in principle — statically provable after the code relying on the truth of the assumption becomes part of a sufficiently large, well-defined cohesive *physical* region. This means, in essence, that within this physical region, all code paths that lead to the location where the assumption is made can be shown to guarantee the truth of the assumption irrespective of any external factors. The physical cohesiveness of the region further connotes that the code encompassing defensive-in-nature checks is collocated with its clients in a manifestly inseparable material way (e.g., within a source file or an executable) and is otherwise physically inaccessible for arbitrary outside use. This physical region is referred to as the check's *neighborhood*.[5]

---

[4] Whether or not an assumption is always expected to be true or is checked and handled by the code, the assumption should always be reflected in the written contract of its associated entity (e.g., a function) as undefined or essential behavior respectively; see **khlebnikov20b**.

[5] Our definition of a neighborhood differs from ostensibly similar definitions that do not involve the aspect of physicality. For example, according to Lisa Lippincott (via private correspondence, February 29, 2020):

> A *logical neighborhood* is a portion of a system that can be reasoned about, understood, and validated independently of other parts of the system. A typical small neighborhood in a C++ program is a function implementation together with its interface and the interfaces to other parts of the system, without which the function implementation cannot be understood [**lippincott16, lippincott19**]. Some neighborhoods, such as the neighborhood of dynamic

Inherently versus Contextually Defensive Checks

The neighborhood of some checks is limited to the source file in which the check is defined (its *immediate neighborhood*). A check that can be proven to always pass for any deployment of its immediate neighborhood, i.e., in any syntactically correct program containing the check, is referred to as *inherently defensive.* Note that when assessing what constitutes an inherently defensive check, only reasonable[6] coding practices are considered.

Inherently defensive checks, despite their limited applicability, are often useful to verify certain programmatic assumptions locally. For example, to double-check the result of an algorithm, an inherently defensive check might use the result of a simpler but perhaps slower or more constrained algorithm implemented in the same source file (see Figure 2).

```
bool is_big(int x) { return x <= INT_MIN / 2 || INT_MAX / 2 <= x; }

int average(int a, int b)
{
    const int res = a / 2 + b / 2 + (a % 2 + b % 2) / 2;

    // Check that when no overflow is possible, this simple,
    // definitional midpoint algorithm yields the same result
    assert(is_big(a) || is_big(b) || res == (a + b) / 2);
    return res;
}
```

Figure 2: Function returning the average of two signed integers employing an inherently defensive check to catch local coding defects early.

A thorough test suite would immediately confirm that the code as written in Figure 2 is actually incorrect (e.g., `a = -2` and `b = 1` produces `-1` instead of `0`), but unfortunately not all developers write sufficiently thorough test drivers. Although runtime checking is not a substitute for proper testing, the inherently defensive check offers a partial verification that helps quickly expose subtle conceptual defects or annoying typos.

More typically, the truth of a check designated as defensive-in-nature cannot be

initialization of a namespace-scope object, are not function neighborhoods.

Neighborhoods typically have a boundary: a portion that cannot be logically separated from the interior of the neighborhood, but also cannot be logically separated from the exterior. The boundary of a function neighborhood is the set of interfaces by which it connects to the rest of the system. We can form — and reason about — larger neighborhoods by gluing neighborhoods together along matching boundaries. (The terms “neighborhood,” “boundary,” “interior,” “exterior,” and “gluing” come from topology; a procedural system can be described as a bitopological manifold [**lippincott18**].)

[6] I.e., responsible, productive, non-malicious. Using the preprocessor to somehow change the meaning of identifiers or modifying the assembly output (post-compilation) would violate the premise of reasonable practice.

proven locally without considering the specific context of its usage. Such *contextually defensive* checks are, however, anticipated to be embedded in larger systems. Within this larger system, each contextually defensive check defines a cohesive neighborhood for the check wherein the check's truth can be proven statically, irrespective of whether that check is performed. In other words, once such proof is provided, the check becomes *manifestly defensive* with respect to its neighborhood.

Contextually defensive checks can be used to verify a much wider range of programmatic assumptions but require a much deeper analysis to prove that they are truly redundant. For example, if an `intLog` function has external linkage and uses a defensive check to verify its precondition,[7] i.e., that the argument is positive and the base is greater than 1, the truth of this check cannot be proven using only the information available in the function's immediate neighborhood. However, once the function is inseparably bound into a cohesive physical unit (e.g., a statically linked executable) with all its callers, the application owner can prove that the check is manifestly defensive by inspecting all the call sites of `intLog` (see Figure 3).

```
// intLog.cpp
int intLog(int arg, int base)
{
    assert(1 <= arg && 1 < base);  🡐 Contextually defensive check
    // ...
}

// log2AbsOrZero.cpp
int log2AbsOrZero(int x)
{
    if (0 == x || std::numeric_limits<int>::min() == x) return 0;
    return intLog(std::abs(x), 2);  🡐 The only call site in the entire application
}
```

Figure 3: A contextually defensive check becoming manifestly defensive with respect to its physical neighborhood of a statically linked executable.

Contextually defensive checks verifying preconditions are particularly beneficial for library development due to their propensity to detect misuse by programmatic clients. In particular, they serve as a valuable and low-cost safety net for application developers who typically have relatively few software clients over which to amortize their development costs, a much wider domain to cover, and (due to direct business drivers) far less time to achieve thorough test coverage.

PRESERVING NEIGHBORHOOD COHESION

The proofs of the assumptions within a physical neighborhood might be rendered suspect if the cohesion of the neighborhood is not preserved throughout all stages of

[7] A precondition is what is required of a function's arguments and of the state of the program (including object state) for a function call to be valid. For a detailed discussion of why specifying a *narrow contract* (i.e., one having preconditions) is often beneficial, see **khlebnikov19a**.

software system delivery — i.e., from building and packaging to use in the intended environment. If neighborhood cohesion cannot be preserved for a particular check (e.g., if manual modification is allowed after system deployment), the check cannot be classified as defensive in nature and should instead be treated as mandatory input validation. Addressing potential disruptions of assumption proofs during system delivery is particularly important for large-scale software systems, which is illustrated below.

PACKAGING. The information sources used by a defensive-in-nature check must be packaged in a physically cohesive unit together with the check itself for assumptions about the input supplied by such sources to be reasonably made. For example, if an assumption relies on a human operator or an unknown remote client of a REST API to always provide data satisfying certain restrictions, then such a source of data has to be packaged into a physically cohesive unit along with the check. Such physically cohesive packaging is, of course, entirely infeasible. Absent a supportive, physically cohesive neighborhood, employing any defensive-checking framework to validate an assumption that relies on such un-vetted input at run time is ill conceived (see Figure 4). In contrast, when all programmatic clients of a function containing a defensive-in-nature check are known to provide valid data and are in some way packaged together within a cohesive physical neighborhood of the check (e.g., a statically linked executable), the (anticipated) use of a defensive-checking framework is justified.

```
int main(int argc, const char *argv[])
{
    assert(2 <= argc); 🡐 BAD IDEA: Asserting number of command-line input arguments
    std::ifstream file(argv[1]);
    assert(file); 🡐 BAD IDEA: Asserting an external file is successfully opened

    // Continue working with 'file'...
}
```

Figure 4: An example of a (poorly engineered) command-line utility that misuses a defensive-checking framework (namely `<cassert>`) to perform input validation.

TESTING. Performing integration tests of a software system as a cohesive packaged unit ensures that the system — as a whole — is assembled properly. The active runtime checking that defensive checks provide complements (but does not substitute for) such integration testing by helping to detect latent defects early and close to their root cause. Violations of defensive checks at this stage might also indicate their incorrect classification as defensive with respect to the physical neighborhood under test.

In addition, repeating integration tests for the software system that was built and packaged with defensive checks disabled is prudent. Doing so allows the discovery of defects in the defensive checks themselves, e.g., exposing checks that contain side effects that affect the program’s essential behavior.[8]

Deployment. Proper packaging and testing of the software system is necessary but insufficient for maintaining the correctness of programmatic assumptions. The deployment strategy must also consistently maintain the physical integrity of the software system. For example, if the software system consists of multiple processes operating together, packaging and testing a new version of all the processes and then deploying them separately might lead to older versions of some processes communicating with newer versions of others. Such version mismatches might violate the notion of a cohesive neighborhood; any proofs for programmatic assumptions would be rendered invalid, and integration tests performed only for the newer versions working together would be irrelevant. If the deployment approach does not allow for simultaneous deployment of the system in its entirety, then the system cannot be considered a cohesive physical neighborhood and communication between the processes is therefore not a viable candidate for defensive checking.

Consumption. Finally, once the software system is deployed, the cohesion of the neighborhood comprising its subsystems must be preserved during the system’s consumption by its clients. Otherwise, programmatic assumptions might be violated. For example, if the client accesses multiple services communicating with each other, the load balancer must ensure service version consistency within the neighborhood.

The PTDC Criteria

In a properly implemented system, the physical neighborhood of every check designated as defensive in nature must remain a cohesive physical unit, starting from packaging and continuing throughout testing, deployment, and ultimately consumption by the intended clients. The packaging, testing, deployment, and consumption criteria are referred to collectively using the initialism PTDC, or *the PTDC criteria*. Ensuring that physical neighborhoods satisfy all four PTDC criteria is essential for long-term stability and robustness of a software system as a whole.

The larger and the more complex the software system, the more diligence required to uphold the PTDC criteria regarding communication between subsystems. The application owner absolutely must make a deliberate decision regarding whether the software system *as a whole* is considered a cohesive physical neighborhood or *which of its constituent subsystems* are. Informing this decision is the engineering tradeoff between the benefits afforded by designating more checks as defensive in nature (e.g., simplified code and higher performance) versus the complexity of the infrastructure required to ensure that the PTDC criteria are satisfied at larger scale. Ensuring that all assumptions about the input originating outside the relevant neighborhood are always verified with essential input validation checks (never with redundant defensive checks or never left unverified) is important from the developer’s perspective.

[8] Properly codifying defensive checks requires avoiding side effects in their predicates. Depending on the specifics of the side effect, however, some of them might be tolerable or even completely benign; see Appendix A for details.

The PTDC criteria can be applied to systems that include more than a single executable, e.g., those involving information arising from other programs, data, tools, and so on. For example, a configuration file read at run time might be deemed to provide consistently and permanently reliable information — and thus be amenable to defensive checking — if the executable along with the configuration file are intended to constitute a container image. To satisfy the PTDC criteria, however, this container image must be treated as a cohesive physical unit in that none of its constituent parts are modified after packaging and that the image is tested, deployed, and consumed in its entirety.

Furthermore, the engineering teams developing multiple, distinct, dedicated services deployed exclusively within a single cluster might deem even network communication among these services as trusted. Hence, the information the services exchange could reasonably be assumed to be valid and verified at run time by checks designated as defensive in nature. Confirming that information external to an executable will be reliable (beyond a reasonable doubt) may, however, involve monumental effort, often requiring complex deployment and system-wide testing, potentially specific to the target hardware. What's more, even the physical hardware must be sequestered within a physically confined and secured area (e.g., a proprietary data center) to preclude data modification by intermediaries. If such effort is not justified (or perhaps if it's even impossible, e.g., due to the services' being accessible to anyone on the Internet), then the communication among even concurrently deployed subsystems cannot be assumed to satisfy the spirit of the PTDC criteria, regardless of the precise mode of communication (e.g., sockets, named pipes, shared memory segments, runtime-loaded shared library, or language bindings). Making such an assumption about the satisfaction of PTDC criteria would be flat-out wrong.

SUMMARY

Use of defensive-checking frameworks is reserved for checks that can be reasonably classified as defensive in nature. Checks whose neighborhoods are not necessarily expected to eventually satisfy the PTDC criteria are not defensive in nature and therefore are ill-suited to such frameworks. The definitions provided above are reprised concisely for convenience in Figure 5.

Contextually Defensive Check — A defensive check that is not inherently defensive, i.e., one that is defensive in nature but whose truth cannot be proven from its immediate neighborhood. Hence, in every case where the check is part of an entity that is consumed by external users, sufficient information is anticipated to always be available (at compile time) to prove (at least in principle) that the check is manifestly defensive.

Defensive Check — A runtime check that is intentionally redundant and inherently optional and that must necessarily be true when incorporated into any defect-free program, system, or other entity that is presented for consumption by external users.

Defensive in Nature — A property of a check whereby the check itself is provided with the understanding that the unit of software implementing that check is either already manifestly defensive (i.e., inherently defensive) or will invariably be bound into a larger entity satisfying the PTDC criteria, which will in turn render the check manifestly defensive.

External User — A consumer of an entity that does not (e.g., cannot reliably) satisfy the PTDC criteria for the entity.

Immediate Neighborhood — The atomic physically contiguous (monolithic) region surrounding the implementation of a defensive check (e.g., the source file) that is devoid of constructs that might reasonably cast doubt as to whether the otherwise non-contextually defensive check is, in fact, manifestly defensive (e.g., conditional compilation or intervening `#include` directives and omitting from consideration wantonly reckless or malicious acts, such as redefinition of keywords).

Manifestly Defensive Check — A check is manifestly defensive for a given physical region if the information contained within that region is sufficient to prove (at compile time and in any build mode) that the assumption it checks is true in *every* context for which that region might reasonably be incorporated for consumption by external users.

Neighborhood — A physical subregion of an entity containing a defensive check that, when embedded in a system satisfying the PTDC criteria, would be sufficient to render that check manifestly defensive.

Inherently Defensive Check — A *non-contextually* (unconditionally), manifestly defensive check, i.e., one whose unconditional redundancy (within *every* syntactically correct program) can be proven locally (e.g., by a human reviewer), irrespective of whether and how its immediate neighborhood is ultimately bound into other entities for consumption by external users.

PTDC Criteria — *Packaging*, *testing*, *deployment*, and *consumption* criteria that the software delivery process of the initially physically separable constituent parts of entity have to satisfy for the proofs of the truth of contextually defensive checks to be reliable within a single, immutable, physically cohesive unit comprising them all.

Figure 5: Summary of terms pertaining to defensive checks.

Real-World Assumption-Checking Scenarios

With a thorough explanation of what conceptually distinguishes defensive checking from input validation completed, a sequence of real-world examples can now be presented. These examples illustrate the range of complexity — from almost obvious to very involved — in properly classifying checks.

Internal Logic Checks

Checks that verify essential properties of implemented algorithms are defensive in nature because, by definition, they are redundant in any defect-free program. For example, such properties may include logic ensuring that an array has been sorted prior to performing binary search, that a certain condition must hold upon exit from a loop, or that a simpler — albeit slower or more constrained (see Figure 2) — algorithm arrives at the same result. Figure 6 illustrates these three checks. Considering that the checks can be shown to hold true using information derived *exclusively* from their immediate neighborhood, they can be accurately classified as inherently defensive.

```
bool containsSamples(const std::vector<int>& data,
                     const std::vector<int>& samples)
```

```
    // Return 'true' if all specified 'samples' are present in the specified
    // 'data' and 'false' otherwise.
{
    std::vector<int> copy(data.begin(), data.end());

    // Sort and remove duplicates
    std::sort(copy.begin(), copy.end());
    copy.erase(std::unique(copy.begin(), copy.end()), copy.end());

    // Internal logic check: 'copy' is sorted and has unique elements
    assert(std::is_sorted(copy.begin(), copy.end()));
    assert(copy.end() == std::adjacent_find(copy.begin(), copy.end()));

    // Do the binary search
    for (int sample : samples) {
        auto   first = copy.begin();
        auto   last  = copy.end();
        size_t count = copy.size();
        while (count > 0) {
            size_t step = count / 2;
            auto   mid  = first + step;
            if (*mid < sample) {
                first = ++mid;
                count -= step + 1;
            }
            else {
                count = step;
            }
        }
        // Internal logic check: 'count' is exactly 0 after the loop
        assert(0 == count);

        bool found = first != last && *first == sample;
        if (!found) {
            // Internal logic check: A linear search yields the same result
            assert(copy.end() == std::find(copy.begin(), copy.end(), sample));
            return false;
        }
    }

    return true;
}
```
Figure 6: Examples of inherently defensive checks.

Unreliable Input Sources

If the source of input for a system is outside of the system, a physical neighborhood cannot reasonably be defined that encompasses both the input and the checks validating it. The PTDC criteria cannot apply to such checks; therefore those checks should not be classified as defensive. The designer should expect that any input may be flawed and prepare to handle such flawed input. Even if the developer intends that the program will abort if it encounters malformed input, this action should not be performed with a defensive check. Rather, the verification should be applied in every build mode.

For example, failing to consistently validate input that might be received from a well-intentioned human operator will inevitably lead to unpredictable intermittent failures. As a second example, input received by a public HTTP server could arrive from a malicious actor aiming to destabilize the system. All such input requests should therefore always be validated thoroughly. Figure 7 illustrates both of these concerns.

```
int main(int argc, const char *argv[])
{
    assert(2 <= argc); 🡐 BAD IDEA: Asserting number of command-line input arguments
    int port = atoi(argv[1]);
    assert(0 <= port && port <= 65535); 🡐 BAD IDEA: Asserting specific command-line values
    HttpServer().listenForever(
        port,
        [](const HttpRequest& request) {
            // ALL checks below are misclassified as defensive. 🡐 VERY BAD IDEA
            assert(request.method() == "GET");
            assert(request.uri() == "/")
            assert(request.headers().content_type() == "application/text");
            assert(request.data().size() <= 1024);

            // ...
        }
    );
}
```

Figure 7: An example of a (poorly engineered) public-facing HTTP server that misuses a defensive-checking framework (namely `<cassert>`) to perform input validation.

Precondition Checks

A function's contract may impose certain *preconditions*, i.e., semantic limitations on syntactically valid inputs or object (or program) state, for its invocation to be considered valid. Failure by the caller to satisfy any one of those preconditions results in (library) undefined behavior, which is automatically considered a software defect, irrespective of whether this results in observably incorrect behavior of the program. The set of preconditions and *postconditions*, i.e., what the function guarantees to have happened given valid arguments and proper state, form a *contract* between the function and its clients.

The client invoking a library function (including one having preconditions) is generally unknown to the function. Nonetheless, the caller and callee are expected to eventually become part of a larger cohesive entity (typically an executable program) that satisfies the PTDC criteria. Classifying precondition checks as contextually defensive and employing a defensive-checking framework to detect inadvertent function misuse by programmatic clients is, therefore, a reasonable practice.[9] For example, a

[9] Note that a function is never under any obligation to (defensively) check all (or even any) of its preconditions that are (or should be) fully documented as part of its (natural-language) contract. Not

binarySearch function extracted from the containsSamples function in Figure 6 might require, as a precondition, that the input range be sorted, as illustrated in Figure 8.

```
// Precondition: [first, last) represents a nondecreasing sequence of values.
bool binarySearch(const int *first, const int *last, int value) {
    // Full (expensive) a priori precondition check (1).
    assert(std::is_sorted(first, last));
    auto cur   = first;
    auto count = last - first;

    while (count > 0) {
        auto step = count / 2;
        auto mid  = cur + step;

        // Partial (inexpensive) precondition check (2).
        assert(*cur <= *mid);

        if (*mid < value) {
            cur = ++mid;
            count -= step + 1;
        }
        else {
            count = step;
        }
    }
    // `count` must be exactly 0 after the loop
    assert(0 == count);

    bool result = cur != last && *cur == value;
    // (Expensive) postcondition check (3).
    assert(result == (last != std::find(first, last, value)));
    return result;
}
```

Figure 8: Binary search function that uses defensive checks for its precondition checks (1) and (2) as well as its postcondition check (3).

In contrast to Figure 6 (where a binary search was performed in the context of another,

only is such a check explicitly *not* part of the function's contract, but, in some cases, such a check might be prohibitively expensive if not impossible. Employing a defensive-checking framework, such as BSLS_ASSERT [**bsls**, **khlebnikov20a**] or the one originally proposed for C++20 [**dosreis18**], facilitates enabling inexpensive checks (e.g., *default* level) without necessarily enabling more expensive ones (e.g., *audit* level). A library developer can then provide a diverse set of clients with better control over apportioning runtime resources commensurate with their own respective states in the software development lifecycle.[10] Another consequence of extracting the binarySearch function is that the full precondition (1) and the postcondition (3) checks become more algorithmically complex than the function itself. This is not an issue in Figure 6 since the overall complexity of the containsSamples function is not increased by defensive checks, but it might lead to assertion-enabled builds becoming unusable for other clients of binarySearch. This complexity highlights the value of partial precondition checking (2) and more sophisticated defensive checking facilities that allow specifying different assertion levels (see **khlebnikov20a**).

larger function) and after factoring out an independently callable `binarySearch` function, the `is_sorted` check, while still defensive in nature, changed its category from an inherently defensive internal logic check to a contextually defensive precondition check. In the context of the original `containsSamples` function, this check is, however, obviously manifestly defensive.[10] This duality reflects both the intuition behind why precondition checks are defensive in nature and also how a change in the physical neighborhood of the check might well affect its classification (i.e., inherently versus contextually defensive).

Resource Files

Checking the validity of external resources, such as configuration files, is typically within the purview of input validation, especially if external users can modify such files. A typical application performing such input validation is illustrated in Figure 9.

```
struct DatetimeIntervalUtil {
    static bool isValidCalendarInterval(const DatetimeInterval& interval);
        // Return 'true' if the specified 'interval' is valid according to the
        // calendar and 'false' otherwise.

    static DatetimeInterval parse(std::string_view data);
        // Parse a DatetimeInterval from the specified 'data'.  The behavior
        // is undefined unless 'data' contains a valid date-time interval.

    static int tryParse(DatetimeInterval *result, std::string_view data);[11]
        // Load into the specified 'result' a date-time interval defined by the
        // specified 'data'.  Return 0 on success, and a nonzero value if 'data'
        // does not contain a pair of formatted valid date-time values or as if
        // 'isValidCalendarInterval' returns 'false' for the parsed interval.
};

int main(int argc, const char *argv[])
{
    if (2 > argc) { std::cerr << "Configuration file not provided."; return 1; }

    std::ifstream config(argv[1]);
    if (!config) { std::cerr  << "Can't open file " << argv[1]; return 2; }

    std::vector<DatetimeInterval> intervals;
    std::string line;
    while (config >> line) {
        DatetimeInterval interval;
        if (0 != DatetimeIntervalUtil::tryParse(&interval, line)) {
```

[10] Another consequence of extracting the `binarySearch` function is that the full precondition (1) and the postcondition (3) checks become more algorithmically complex than the function itself. This is not an issue in Figure 6 since the overall complexity of the `containsSamples` function is not increased by defensive checks, but it might lead to assertion-enabled builds becoming unusable for other clients of `binarySearch`. This complexity highlights the value of partial precondition checking (2) and more sophisticated defensive checking facilities that allow specifying different assertion levels (see **khlebnikov20a**).

```
            std::cerr << "Invalid date-time interval encountered.";
            return 3;
        }
        intervals.push_back(interval);
    }

    // Continue with valid 'intervals'...
}
```

Figure 9: Properly validated (user-supplied) configuration.

However, business drivers might exist that could motivate changes in the software design and subsequent recategorization of certain checks. For example, if the code in Figure 9 were the startup routine of a microservice, the developer might want to reduce the startup time by avoiding complex (due to calendar access) checks of the interval validity performed by `tryParse` in a tight loop. The developer would first benchmark the startup code to confirm that removal of these validity checks would lead to significant improvements. Afterward, the developer might choose to use the `parse` function, whose behavior is undefined if textual representation of an invalid interval is supplied to it. Such change is equivalent to recategorizing the assumption of data validity from one that might be false to one that is always expected to be true.

In this design, the contents of the configuration file would become part of the neighborhood of contextually defensive checks in the `parse` function. Therefore, the software system encompassing the executable and the configuration file must satisfy the PTDC criteria. This entails more than defining a fully encapsulating process (e.g., one that uses containerization). Satisfying the PTDC criteria also involves the installation and enforcement of policies that dictate that changes to constituent parts of the system necessarily restart the software delivery process. For example, the software delivery process must be restarted if even the most qualified and careful engineer makes manual changes to the container image. Figure 10 illustrates how such a containerized application might be implemented. Note that despite the container image having full control over the command-line arguments, recategorizing the checks that perform their simple validation as defensive in nature serves no business purpose and is not justified.

```
int main(int argc, const char *argv[])
{
    if (2 != argc) { std::abort(); }  // No business drivers to convert to assert.

    std::ifstream config(argv[1]);
    if (!config)   { std::abort(); }  // No business drivers to convert to assert.

    std::vector<DatetimeInterval> intervals;
    using string_it = std::istream_iterator<std::string>;
    std::transform(
        string_it(config), string_it(),
        std::back_inserter(intervals),
        &DatetimeIntervalUtil::parse);  // Contents defensively checked by 'parse'.
    // Continue with valid 'intervals'...
}
```

Figure 10: Containerized application with configuration file contents check reclassified as defensive.

Automatic safeguards against resource file modifications can complement and, in some cases, even replace containerization while fully satisfying both the letter and the spirit of the PTDC criteria. For example, one can checksum the data in the file using a secure hash, such as SHA-2 (e.g., SHA-256), and embed that in the source code of the program. Then, when the file is read, its checksum is unconditionally verified against the embedded hash using a conventional `if` statement. If the checksums match, the program proceeds normally; otherwise, a short, descriptive message is printed and the program explicitly exits, e.g., using `std::abort()`.

Defining Neighborhoods in Distributed Computing

Thus far this paper has considered defensive checks in entities as small as the body of a single function to systems comprising programs, files, and other artifacts executing on a single computer. Given our strict criteria for employing defensive-in-nature checks, using them to (defensively) check data passing across process boundaries (let alone among processes running on multiple machines) might seem dubious. However, defensive checks can be viable in neighborhoods larger than those previously considered. Their applicability, as ever, is enabled by physical proximity and governed by compliance with the PTDC criteria.

Shared-Memory Multi-Process Systems

Consider a request/response system running on a single multicore supercomputer that handles concurrent users by spawning numerous identical processes. In this system, the state of each active client session can be kept in static memory, swapped out (in binary form) to secondary storage using memory-mapped I/O, and then later swapped in again (at the same virtual memory address) to any of the available processes. What makes such an architecture feasible is the presumption that each of the processes are identical clones; if so much as a single byte in a relocatable image of one of the processes were to diverge, the save/restore functionality would likely fail, often spectacularly. Given a robust library that defensively checks object invariants,[12] enabling those checks when designing the infrastructure to spin up these processes might be very useful in detecting defects. Once that mechanism is sufficiently proven, the system owner may eventually choose to disable those invariant checks while making no other changes to the system. In a defect-free system, all object invariants

[12] An *object invariant* is an assumption that, in every defect-free program, is true from the moment an object's constructor returns until the moment that object's destructor is invoked. The only exception is, perhaps, during execution of one of that object's member or `friend` functions (i.e., any function having access to that object's non-public state).[13] **khlebnikov19b** is a conference talk that inspired this paper.

will hold irrespective of whether they are checked (redundantly) at run time; hence, the neighborhood of these invariant checks satisfied the PTDC criteria.

Distributed Memory Systems

Consider two computers that communicate via sockets using, say, the HTTP/2 wire format. Is it ever reasonable to presume that the information traveling between these computers satisfies the PTDC criteria? As previously stated, if the sockets are connected using a public network, then the answer is an emphatic no. If, however, the connection is via a dedicated line and the computers are sequestered (e.g., confined to a single, secure room) and controlled together (i.e., under the same authority), then the entire room might reasonably be treated as a cohesive neighborhood. Of course, defensive checks in this neighborhood rely on it satisfying the PTDC criteria and on no part being subject to independent modification after the system has been secured.

As the final example, consider a large data center, such as might be found at a major financial information services company. These massive computing facilities contain a large number of server machines that must run continuously 24/7 and have no opportunity to be stopped, updated, tested, and redeployed in unison. Yet, at this scale, eliminating even a small percentage of the cycles per customer request can translate to significant savings in terms of reduced hardware, heat dissipation, and so on.

For illustration purposes, consider a fairly small computing center consisting of just 100 machines, arranged in a ten by ten array, **`M[10][10]`**. Each machine spawns roughly 1000 nearly identical processes, amounting to a total of 10^5 processes across the computing center. Each time this massively replicated process is to be updated, all of the relevant, fully unit-tested componentized software is linked to form a physically monolithic executable image, which, now immutable, is then beta-tested in a simulated production environment. Such simulated production testing is very valuable but is no substitute for production hardening, so eventually the software will need to be exercised by live customers in production.

Deploying new software, unit-tested or not, to production is a delicate task and must be done carefully. Hence, a new version of the server process is never rolled out to user machines all at once but in increasing (e.g., quadratically) waves. First, a single user (server) machine is brought down, say **`M[0][0]`**. The current version of the executable is replaced with the new one, and all the processes on machine **`M[0][0]`** are spun up. The machine is then brought online and developers basically wait to see what happens. If, after some time, no problems arise, the rollout is continued by bringing up, say, three more machines, e.g., **`M[1][0]`**, **`M[1][1]`**, and **`M[0][1]`**, and again the developers wait. With each successive wave, more machines are rolled out until all of them are executing processes spawned by the same new executable. If at any point a problem manifests, then the process is reversed to return the system to its previous state. So, how can defensive checks help us here?

In the absence of defensive checks, the rollout process must proceed slowly because, unless the system crashes outright, the (human) customers typically require time to observe anomalies, realize things aren't quite right anymore, and call customer support to report the newly experienced problem. Now suppose instead that each new system was built in two ways: with and without defensive checking enabled. How might the process be different? It might start at deployment of the slower but more robust defensively checked version on **`M[0][0]`**. With defensive checking enabled throughout every process on **`M[0][0]`**, any violations of defensively checked internal assumptions (i.e., those relating specifically to the correctness of the process itself) will be quickly flagged and the next wave can be aborted much earlier. If, after a short delay, no such problem is reported, the process can continue (much more quickly than without defensive checking enabled) with the second wave and so on.

Since defensive checking requires additional computational resources, subjecting all available machines to performing such redundant runtime checking for an extended time is undesirable. As the first wave continues to spread over the example computer farm, a second wave that replaces the defensively built executables with leaner, non-defensive ones can be started, and developers will have greatly reduced concern that these new, higher performance executables will be disruptive in production.

Two concepts make this approach work reliably:

1. The executable itself satisfies the PTDC criteria and is then tested as a non-modifiable unit before it is ever deployed to production.
2. Once all of the new executables are deployed and running in unison, they too have effectively been assembled as a yet larger, non-modifiable unit satisfying the PTDC criteria. The entire computer farm can, in effect, be considered one gigantic neighborhood.

Hence, once everything appears to be working well, enduring the often substantial runtime overhead of always rechecking what now satisfies the PTDC criteria, is provably correct (in principle), and is observably so (in practice) is now unnecessary.

Additionally, because all the processes are, by design, identical and running on similar hardware, any one of them can serve as a safeguard to sample the client traffic of the server farm. So, instead of removing all of the 100 defensively instrumented executables from the farm, leaving a few machines running the slower, more robust, defensively checked version in place might be beneficial. In this way, statistically significant, practically useful defensive checking can be performed at a moderate cost on a random subset of customer queries — from 100% to 1% or anything in between — just in case something in the external environment changes such that previously unproven code paths begin to execute.

Finally, this dual-wave–based rollout approach naturally scales to computer facilities of almost any size. Instead of having just a two-dimensional 10 x 10 grid, consider a three-dimensional block of machines **`M[100][100][100]`** in an edifice the size of a warehouse (10^9 processes). The same sort of two-phased, multi-wave rollout approach (while keeping just a tiny fraction of the machines enabled for statistically useful defensive checking) pertains.

Example Summary

This paper has presented a series of real-world examples highlighting various principles for selecting defensive checking versus input validation. Figure 11 presents a concise summary of the principles elucidated in each of the examples.

Subsection Title/Topic	**Principles Being Demonstrated**
Internal logic checks	Immediate physical neighborhood Inherently defensive checks
Unreliable input sources	External users Input validation
Precondition checks	Contextually defensive checks PTDC criteria Manifestly defensive checks
Resource files	Evolving physical neighborhood
Larger neighborhoods	PTDC criteria satisfied through replication Staged rollout with defensive checking Statistically significant partial checking

Figure 11: Brief summary of principles elucidated per subsection.

CONCLUSION

Making assumptions is inherent to writing any software system. Two distinct and non-overlapping kinds of assumptions have been identified:

(1) those pertaining to the correctness of a software system itself (i.e., the system does what it is expected to do)
(2) those pertaining to the validity of the external data passing across autonomous system boundaries (i.e., the externally supplied input conforms to what the system is expected to handle).

Assumptions of the first kind are generally *knowable* and (in principle) provable based solely on the information available when packaging the system or subsystem. Hence, any subsequent runtime validation of such assumptions is redundant, entirely superfluous in a defect-free program, and referred to generally as *defensive checking*. Assumptions of the second kind, on the other hand, are *unknowable* locally; must (for correctness) always be validated at run time; and, whenever determined to be false, must somehow be handled (even if only to reliably terminate execution). This second category of assumption checking, referred to generally as *input validation*, must always continue to be present and active (e.g., in *every* build mode).

Consistently discriminating accurately between these two disjoint assumption categories is critically important for software systems to be correct (and thus stable). Failing to properly categorize an assumption, which can (and often does) happen in practice, might lead to both inefficiencies (e.g., when a defensive check outlives its usefulness) and catastrophic failures (e.g., when checks that are required even in an otherwise defect-free program are inappropriately disabled in the name of runtime performance).

This paper has identified several important properties related to successfully characterizing whether the truth of a given assumption will ultimately be knowable before run time or will always require runtime validation (see Figure 5 for a taxonomy). A check is *defensive in nature* if its truth is — or is anticipated always to be — statically provable from information proximately available in some well-defined physically cohesive region called a *neighborhood*. Each defensive-in-nature check is *inherently defensive* if it can be proven irrespective of the context in which its (inherently physically contiguous) *immediate neighborhood* resides; otherwise, it is *contextually defensive*.

Each contextually defensive check should (in principle) be statically provable for a certain cohesive physical neighborhood within the software system, i.e., it should become *manifestly defensive* with respect to this specific neighborhood. For such proofs to be reliable and robust throughout the software delivery process, however, the neighborhoods comprised of physically separable entities have to satisfy the PTDC criteria. Such deliberately identified physical neighborhoods afford the adjudication of whether a check can be considered defensive in nature and, hence, whether the use of a defensive-checking framework, such as `<cassert>`, for the check's implementation is justified.

This paper posits that even properly categorized defensive checks are no substitute for thorough unit testing but are effective at accelerating code-defect detection — both during development and after deployment to production. Moreover, defensive checks in library code provide a welcome safety net for application clients, especially when inevitable time pressures preclude a more methodical and systematic (e.g., unit-testing) approach. Even proofs that supposedly cannot be wrong (in theory) occasionally are (in practice). The redundancy of (sometimes) calculating something in two very different ways and getting the same result adds a solid measure of confidence that the calculation is correct. As implementers of defensive checks, however, developers must always be mindful that some assumptions are inherently defensive in nature while others are not.[13]

References

bsls. https://github.com/bloomberg/bde/tree/master/groups/bsl/bsls

dosreis18. G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup "Support for Contract Based Programming in C++," *C++ Standards Committee Working Group ISOCPP*, Technical Report P0542R5, 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html

khlebnikov19a. R. Khlebnikov and J. Lakos. "Contracts, Undefined Behavior, and Defensive Programming," *C++ Standards Committee Working Group ISOCPP*, Technical Report P1743R0, 2019 (originally published internally to Bloomberg, 2017).

[13] **khlebnikov19b** is a conference talk that inspired this paper.

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1743r0.pdf

khlebnikov19b. R. Khlebnikov "Avoid Misuse of Contracts!" *C++ Conference* (CppCon), Aurora, CO, September 2019.
https://youtu.be/KFJ5p-T-S7Q

khlebnikov20a. R. Khlebnikov and J. Lakos. "Defensive Programming using `BSLS_ASSERT/BSLS_REVIEW`," *C++ Standards Committee Working Group ISOCPP*, Technical Report Draft D2110, forthcoming.

khlebnikov20b. R. Khlebnikov and J. Lakos. "Delineating C++ Contracts in English," *C++ Standards Committee Working Group ISOCPP*, Technical Report Draft D2111, forthcoming.

lakos20. J. Lakos. *Large-Scale C++ Volume I: Process and Architecture*. Boston: Addison-Wesley, 2020 (published December 17, 2019).

lippincott16. L. Lippincott. "Procedural Function Interfaces," *C++ Standards Committee Working Group ISOCPP*, Technical Report P0465R0, 2016.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0465r0.pdf

lippincott18. L. Lippincott. "The Shape of a Program," ACCU, Bristol, April 2018.
https://youtu.be/IP5akjPwqEA

lippincott19. L. Lippincott "The Truth of a Procedure," *C++ Conference* (CppCon), Aurora, CO, September 2019.
https://youtu.be/baKqCOLKcPc

Appendix A: Characterizing Side Effects in Defensive-Checking Predicates

A proper defensive check must satisfy two requirements:

(1) In a defect-free program, the check must pass.
(2) Removing any given check must — independent of any other such check — have no effect on the essential behavior of the software.

While these two requirements are closely related, satisfying the first without satisfying the second is possible, most typically by erroneously allowing a side effect that contributes to essential behavior to be a part of the predicate of a defensive check. For example, in Figure 12, in an attempt to verify that the value supplied to the `addField` method is successfully inserted into an `std::map`, the call to `emplace` appears as the predicate of an `assert` statement. If this code is compiled with `-DNDEBUG`, the value will not be inserted at all, thereby altering the essential behavior of the software.

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    void addField(std::string_view name, std::string_view value)
    {
        assert(d_fields.emplace(name, value).second);
    }
};
```

Figure 12: Example of an essential side effect incorrectly used in the predicate of a defensive check. Correct code would place the return status in a variable and assert its value in a separate statement.

As it turns out, however, not all side effects are equally problematic. Depending on the software requirements, some side effects, such as print statements, temporary memory allocation/de-allocation, or even (persistent) logging, may be allowed (or at least tolerated) in a defensive check's predicate because essential behavior is unaffected. A side effect in a defensive check's predicate is tolerable if the presence or absence of the side effect in any given thread of program control has no effect on the essential behavior of the program. A side effect in a defensive check's predicate within a given code path is benign if that side effect can have no effect on nonlocal (i.e., any other) observable behavior within the program.[14] Under these definitions, an alternative (valid) implementation of the `addField` method (from Figure 12) might incorporate such benign or tolerable side effects in its defensive checks as illustrated in Figure 13.

```
class HttpHeaderFields {
    std::map<std::string, std::string> d_fields;

  public:
    bool contains(std::string_view name) const
    {
        std::cout << "Checking whether '" << name << "' is present among "
                  << d_fields.size() << " fields.";  // (1)

        return d_fields.find(std::string(name)) != d_fields.end();  // (2)
    }

    void addField(std::string_view name, std::string_view value)
    {
        assert(!contains(name)); // (3)
        d_fields.emplace(name, value);
    }
};
```

Figure 13: Examples of both (1) benign *console output and (2)* tolerable *temporary allocation* side effects *used in a (3)* defensive-check predicate.

[14] A side effect that is not legitimately observable programmatically is not considered a side effect for the purposes of this discussion. An example of such a side effect is calling a function that might alter bits on the program stack in a manner that cannot be accessed without triggering undefined behavior or one whose only effect is that it takes longer and/or dissipates more heat,.

APPENDIX B: PRECONDITION CHECKS IN HIERARCHICALLY REUSABLE LIBRARIES

During the development of hierarchically reusable software,[15] a piece of low-level functionality is, not uncommonly, used locally in other functionality where its precondition checks are initially inherently (and hence manifestly) defensive, and then later, after fine-grained physical factoring, only contextually defensive, as evidenced in Figures Figure 6 and Figure 8, respectively. Another common practice is to expect that a particular assumption regarding a reusable function's arguments and/or an ambient object's (or program's) state might naturally be able to be guaranteed in some calling contexts. A check for this assumption would thereby qualify as (contextually) defensive (often with no need to return status). Other clients, however, might be better served if this assumption were addressed in the input-validation realm, with the function always checking and reporting a failure status whenever the assumption is false.

Having just a contextually defensive check would force all clients to perform the check themselves, even if that might mean duplicating work that will need to be done anyway. Providing only a (permanent) validating check would impose an unnecessary performance penalty on all clients that can themselves guarantee, at little or no added cost (or risk of coding error), that a function's preconditions are satisfied. Empowering the library client to decide whether their particular use case requires (optional) defensive checking or (essential) input validation is, therefore, prudent.[16] This sort of pseudo-dual (defensive versus input checking) classification can be approximated by allowing a single function to be configured via a runtime flag (or at compile time using a function template parameter). Nonetheless, providing two entirely distinct functions — each customized to suit its respective client's manifestly different needs — is almost always wise. Figure 14 illustrates one way of rendering such a dual API supplemented by a validity-checking function.

```
struct DatetimeIntervalUtil {
    static bool isValidCalendarInterval(const DatetimeInterval& interval);
        // Return 'true' if the specified 'interval' is valid according to the
        // calendar and 'false' otherwise.

    static DatetimeInterval parse(std::string_view data);
        // Parse a DatetimeInterval from the specified 'data'.  The behavior
        // is undefined unless 'data' contains a valid date-time interval.

    static int tryParse(DatetimeInterval *result, std::string_view data);[17]
```

[15] A *hierarchically reusable library* is designed for general use where each function exposes its fully factored implementation as a fine-grained (acyclic) physical hierarchy of homogenous atomic physical entities called *components*; see **lakos20**, sections 0.4–0.5, pp. 20–43.
[16] Control of whether to perform input validation *must* be entirely in the hands of the immediate client of the reusable library and *must not* be conflated with the global (e.g., build-system level) controls for activating or deactivating defensive checks.

```
        // Load into the specified 'result' a date-time interval defined by the
        // specified 'data'.  Return 0 on success, and a nonzero value if 'data'
        // does not contain a pair of formatted valid date-time values or as if
        // 'isValidCalendarInterval' returns 'false' for the parsed interval.
};
```

Figure 14: Example rendering of a dual API (non-validating alongside validating).

Appendix C: Undefined Behavior in Internal Logic Checks and Postconditions

Whether we consider a postcondition to be contextually or inherently defensive is perhaps of only academic interest since every postcondition is always contractually predicated on *all* of its preconditions being met. The same can be said of any internal logic checks that depend on preconditions being satisfied. Again, to consider a check inherently defensive would require theoretically no syntactically valid way in which that function could be invoked that would produce a result that violates the checked assumption. For consistency, we say that a postcondition along with any intermediate checks in the body of a function can be considered inherently defensive only if (1) the function has a wide contract or (2) the check can otherwise be proven to be true irrespective of any combination of precondition assumptions being met.

In practice, however, internal logic checks and postconditions are routinely allowed to presume that all preconditions are met. This presumption is natural and intuitive given that the code itself makes the same sorts of presumptions in a way that the compiler is free to observe. If, for example, a precondition of a function (e.g., `strlen`) is that a supplied pointer must hold the address of a null-terminated string (and hence is not itself null), then the implementation of the function can reasonably and properly presume (unconditionally) that the supplied pointer is not null and can dereference it without any attempt at validation, since any such (permanent) validating check would be considered supererogatory runtime overhead. Adding here a contextually defensive check for a null pointer cannot introduce new undefined behavior because the very same undesirable behavior will occur regardless of whether the check is active.

When potential undefined behavior is introduced by the predicate of a defensive check, we may choose to guard that implicit assumption with the predicate of a separate (e.g., contextually defensive) check (itself introducing no undefined behavior) that necessarily precedes the ostensibly problematic check in every build mode where it might be active. When there is no possibility that any (language) undefined behavior is introduced by the predicate of a (e.g., defensive) check in any build mode, we refer to such a check as being undefined-behavior-safe, or *UB-safe*. It remains an open question as to whether making *all* defensive checks UB-safe is a best practice, especially when they would otherwise be shadowed anyway.