P2053R0
2020-03-02

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

Defensive Checking Versus Input Validation

NOTE: This whitepaper is intended to advise continued progress toward developing an appropriate, effective, and successful specification for a (runtime) contract-checking facility in the C++ Standard, the specific features of which are largely orthogonal to the thesis of this paper.

N.B.: This paper, the premium version, is rich in details, tangential asides, and examples. A light version of this paper is planned for practitioners who have no need for such pettifogging.

Abstract

For a software system to function as intended, the assumptions made by its designers must be satisfied. Such assumptions fall into one of two distinct categories: (1) The system must be free from observable defects, and (2) the external input entering the system must conform to that system's specifications. Empirical evidence indicates and anecdotal observations support that these two categories of assumptions are often confused and/or conflated in practice, leading to both reduced efficiency and potentially catastrophic failures. In this paper we elucidate important differences between these two categories affecting how runtime checking of these assumptions is implemented properly. We then provide novel criteria based on *neighborhoods* to discriminate between the two. Only those checks that (1) are inherently *defensive in nature* and (2) are or become *manifestly defensive* when deployed can be considered truly redundant in theory and safely removable in practice.

Introduction

The vast majority of software systems accept, in one form or another, certain structured data as input, process that data in accordance with their respective specifications, and produce result data as their output. This data-transformation process may be disrupted in two disparate ways: (1) the software system itself might contain defects thereby preventing proper data processing and (2) the input data might be malformed — i.e., syntactically or semantically inconsistent with the system's specifications. From the perspective of the developer, these disruptions are fundamentally and consequentially different.

1) The developer is in control of the system (e.g., the source code) and can take measures to avoid defects, whereas the input data supplied by the client cannot be so controlled.
2) Over time, the system's defect rate (e.g., newly discovered software bugs) typically decreases, whereas the likelihood of the system encountering malformed input remains largely the same. In today's security-aware world, the need to validate external inputs never goes away.

A well-designed software system will often contain runtime mechanisms for both verifying its own correctness and validating its external input. The correctness checks "defend" against the inevitable defects introduced during software development and subsequent maintenance. Such *defensive checks* are entirely redundant to the functional specification of a (defect-free) software system — i.e., they have no effect on its *essential behavior*.[1] Hence, purely defensive checks can potentially be removed (e.g., 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 perimeter of the trusted region of a subsystem satisfies the requirements imposed by that subsystem. As with every aspect of essential behavior, we expect input validation to be tested thoroughly — e.g., by supplying a wide variety of invalid inputs during unit testing. Once released to production, however, improper input is typically likely to be encountered in roughly equal measure as the system itself hardens — that is, of course, unless the external environment experiences relevant change. These input-validation checks are, therefore, a necessary part of the program's essential behavior and hence cannot (ever) be removed from the final product.

Owing to consequentially prodigious differences between defensive checks and input validation, conflating implementations of these two kinds of developer-assumption checking could easily render the software unfit for its purpose. For example, using a defensive-checking framework, such as `<cassert>`, to implement input validation might allow malformed input to harm the system in a build mode where defensive checks are disabled, thereby leading to hard-to-diagnose crashes, incorrect output, and perhaps even vulnerability to malicious attacks. Similarly, always attempting to validate (and, if defective, circumnavigate) at run time the internal logic of a program (e.g., using hard-coded, unconditional checks) can be just as costly and problematic for very different reasons despite perhaps at first appearing to be less detrimental.

- Attempting to recover from a program defect (and to continue normally after one has been detected) is a dubious engineering practice: Even with code that was carefully and specifically crafted to detect and handle a defect, stipulating that the program would have to fail for the detection and recovery to work as intended implies that same known-to-be-broken program simply cannot reliably uncover the true source and impact of the defect at run time.[2]
- Since the additional code for handling a defect is (by design) unneeded unless the program is defective, this already dubious code acts like an invariant check, is very difficult (if not intractable) to test, and is typically never executed; hence, the code for handling a defect is itself especially likely to contain latent defects.
- Defect handling (i.e., attempting to *recover* from defects as opposed to merely

---

[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**.

[2] Note the practical distinction between a defensive check in newly minted software and a new defensive check in battle-hardened software: While it would be foolish to proceed after the failure of the former, very practical reasons exist for proceeding (reporting and continuing) after the failure of the latter.

detecting and reporting them) affects not only functions that perform the checks, but also *all* of their callers, resulting in a combinatorial explosion of failure modes and greatly increased code complexity generally, thus leading to a profoundly undesirable maintenance burden (see Figure 1).

- Finally, since these defensive checks are implemented with no consideration of whether they might become redundant, they will be present even in a defect-free program, incurring an often noticeable, occasionally substantial, and eventually needless performance penalty in perpetuity.

```
float min_float(const float *begin, const float *end)
{
    if (nullptr == begin)          throw std::invalid_argument{"null begin"};
    if (nullptr == end)            throw std::invalid_argument{"null end"};
    if (std::less<>{}(end, begin)) throw std::invalid_argument{"bad range"};
    if (begin == end)              throw std::invalid_argument{"empty range"};

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

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

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

Figure 1: Significant additional complexity introduced by attempting to handle *(rather than merely detect) software defects. Providing the code needed to handle defects in this function implies that callers of this function also handle all of these logic-error exceptions, vastly increasing their respective complexities as well.*

Despite important differences between these two kinds of checks, developers commonly treat them interchangeably due to their superficial similarities. This is not merely an academic concern: Empirical data from extensive usage within Bloomberg[3] confirm that these misjudgments occur all too frequently in practice with sometimes catastrophic results.

This paper aims to help the reader (1) appreciate the fundamental differences between these two distinct categories of assumptions, (2) learn essential definitions and criteria to be used in discriminating between the two, and (3) apply these discrimination techniques to a series of increasingly nuanced use cases that are typical of real-world production software. Armed with this invaluable knowledge, the reader will begin to develop an intuition for (and, soon, a deep understanding of) how to interpret properly

[3] Stored as part of Bloomberg's internal defect reports. We have no quantifiable data from outside Bloomberg, but we have no reason to believe that it would be substantially different.

sometimes-subtle, real-world situations and when (with relative certainty), where, and why to use which category of assumption check. Although we stop short of describing how to implement defensive (and other) checks using available tools, references to such tutorial materials are cited throughout.

DEFENSIVE CHECKING

A *defensive check* is *redundant* code that is executed at run time to verify that an assumption that must be true in a defect-free system holds at the time the check is executed. Because defensive checks invariably verify programmatic assumptions that are in some sense *local* (see the "Discrimination Criteria" section of this paper), such as a loop invariant, their failure always indicates a software defect. Adding or removing any proper defensive check — e.g., by modifying the source code or, more conveniently, by altering the build mode — has no effect (apart from run time) on the essential behavior of a correct component, subsystem, program, or system. Such redundant checks, however, are invaluable during development (including during unit testing) due to their propensity to expose defects early and in proximity to the source of the error.

Despite their ability to uncover software defects, defensive checks are not by themselves a substitute for proper testing of library code, but they supplement such testing by expediting defect discovery in the code under test. Furthermore, application developers (compared with low-level–library developers) 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; for these developers in particular, having a robust library with defensive checks enabled affords an invaluable and low-cost safety net.

Performing/Codifying Defensive Checks

Properly implementing a defensive check typically involves employing a facility, such as `<cassert>`, that is designed specifically for this purpose.[4] Such defensive-checking facilities typically provide a (i.e., at least one) construct that accepts a boolean predicate and has no effect if the predicate evaluates (or would evaluate[5]) to true; otherwise some alternative action is taken. Typical defensive-checking frameworks will provide ways to control, at compile time, whether a given check is to be active at run time; if such a check is to be inactive, then no runtime code associated with that check will be generated. Notably, defensive checks can always be enabled or disabled externally to the source code (e.g., via build options) — e.g., compiling with `-DNDEBUG` disables C-style `assert` macros and invoking the Python interpreter with the `-O` switch disables Python `assert` statements.

---

[4] Bloomberg has had a proprietary macro-based library facility in place since 2004, which it published it as open source software c. 2010 **[bsls]**. See also **khlebnikov20a** for a detailed review of the BSLS_ASSERT facility.

[5] Depending on the framework, if the predicate expression can be determined to be true at compile time, then evaluating it at run time may be unnecessary.

Proving — solely from (physically) locally accessible information — that a check is entirely redundant and can be eliminated with no change whatsoever to the essential behavior of software itself *nor any potential client thereof* must be possible for a check to be manifestly defensive (see the "Discrimination Criteria" section). For example, Figure 2 illustrates a function that is intended to return *safely* (i.e., with no possibility of overflow) the average, defined as `(a + b) / 2`, of any two signed integers `a` and `b`. The function properly avoids overflow in the general case by distributing the division and then taking supplementary action only if both `a` and `b` are odd.

```
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 a (purely and manifestly) 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, b = 1` produces `-1` instead of `0`), but not everyone writes thorough test drivers (and shame on those who don't). Still, if whatever code we finally wind up with works the same as the simple average for the middle half of the integers and if we can (visually) convince ourselves that the algorithm works at the boundaries of the integer range, then this defensive check is a fairly promising bet to catch subtle conceptual defects or annoying typos. This defensive check also (quickly) exposes the error (i.e., whenever the two inputs have opposite signs, the negative one is even and the positive one is odd).

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, and (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 3, 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 3: 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 defensive-check predicates because essential behavior is unaffected. A side effect in a defensive-check's predicate is *tolerable* if 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.[6] Under these definitions, an alternative (valid) implementation of the `addField` method (from Figure 3) might incorporate such benign or tolerable side effects in its defensive checks as illustrated in Figure 4.

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

[6] A side effect that is not legitimately observable programmatically, such as 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, is not considered a side effect for the purposes of this discussion.

INPUT VALIDATION

Unlike defensive checks, input validation is concerned with verifying whether the data coming into a software system from across its boundary is well formed and suitable for processing. An *input-validation check* is *essential* code that performs a runtime check used to ensure that data entering from outside the perimeter of the trusted region of a subsystem satisfies the requirements imposed by that subsystem; such checks typically provide for a failure path other than merely reporting the error loudly and aborting the process immediately.

Given that even a defect-free program or software system is capable of encountering malformed input at any stage in the software development lifecycle, input validation simply cannot ever be removed from a subsystem safely, i.e., without compromising the correctness (with regard to the essential behavior) of that subsystem.

Naïvely employing mechanisms intended for defensive checking to do input validation, albeit enticingly convenient, is woefully ill conceived and can lead to both (1) insufficient input validation, e.g., when (essential) input-validation checks are disabled in some build modes (which is itself a software defect), and (2) the inability of application owners to disable *any* defensive checks for fear that needed input-validation checks will be disabled as well.

Performing/Codifying Input Validation

Given that input validation is always essential to meeting systems specifications (irrespective of the maturity of the software implementing it), such validation must be performed in all build modes. Use of any form of defensive-checking framework would introduce valid security concerns — especially when the software might be deployed in a manner that could grant unprotected access to a bad actor. Hence, a regular control-flow mechanism, such as an `if` statement, is both suitable and appropriate for this purpose, whereas a strictly defensive check, such as a C-style `assert`, is not.

Alternatively, input validation might reasonably be delegated to, say, a library function provided that the function's (programmatic) API is designed specifically to accommodate arbitrary input.[7] Figure 5 illustrates proper input validation (e.g., using `if` statements and a validating library API) as well as a common mistake (e.g., supplying raw input to the programmatic interface of a nonvalidating API).

```
class ValueCollection {
    // [...]

    static bool isValid(std::string_view value);
        // Return 'true' if the specified 'value' is valid, and 'false' otherwise.

    void add(std::string_view value);
        // [...] The behavior is undefined unless 'isValid(value)' returns 'true'.
```

[7] Such function APIs are said to have a *wide contract* (i.e., they impose no *preconditions* on syntactically valid inputs); see **khlebnikov19a.**

```
    int addIfValid(std::string_view value);
        // [...] Return 0 on success, and a nonzero value otherwise. [...]
        // This method has no effect if 'isValid(value)' returns 'false'.
};

int main(int argc, const char *argv[])
{
    if (2 > argc) {  // (1)
        return 1;
    }

    ValueCollection values;

    values.add(argv[1]);  // (2) ← BAD IDEA

    if (ValueCollection::isValid(argv[1])) {  // (3)
        values.add(argv[1]);                  // (4)
    } else {
        return 2;
    }

    if (0 != values.addIfValid(argv[1])) {  // (5)
        return 2;
    }

    // [...]
}
```

Figure 5: Examples of correct and incorrect input validation: (1) correctly uses a regular `if` *statement for input validation; (2)* improperly *uses a library function having a nonvalidating API; (3) properly validates input before passing it to the nonvalidating API in (4); and (5) optimally uses the validating API created specifically for such usage scenarios.*

SUMMARY SO FAR

We propose that programmers face two distinct categories of assumptions when designing software systems. The first kind of assumption pertains to the correctness of the software system itself, which is ostensibly under the programmers' control — i.e., that the system as a whole works as intended and is otherwise free of defects. The second kind of assumption pertains to the validity of input originating from outside the boundaries of the trusted part of the system — i.e., that such input always satisfies the requirements imposed by the system's specifications and that invalid input is always handled appropriately.

Defensive checks, which check assumptions from the first category, are inherently redundant and optional; the value in performing them typically declines as the software matures. *Input validation*, which validates assumptions of the second kind, is inherently essential and always required; performing it is vital irrespective of the maturity of the software involved.

Facilities designed to perform defensive checks are ill suited to performing input

validation and vice versa. Moreover, due to the assumption categories' superficial similarities, well-intentioned developers can easily (and, in practice, commonly do) conflate these two categories and fail to implement proper checks for these respective assumption categories in production code.

In the following sections, we continue to build upon these general observations by providing a suite of precise definitions and principles pertaining to defensive checks. We then apply these principles to a series of increasingly nuanced real-world examples, thereby better elucidating for the working programmer the sometimes-subtle clues that distinguish defensive checking from input.

Discriminating Between Defensive and Essential Checks

Determining whether checking a given assumption about a subsystem can be properly classified as (at least potentially) redundant and therefore (perhaps at some point) optional, rather than inherently essential and therefore always mandatory, will inform the developer of whether the use of *any* defensive-checking framework (e.g., `<cassert>`) is viable.

With suitable definitions and proper design requirements to guide us, this decision will often be straightforward, yet, in many real-world scenarios, implementing such a check appropriately (let alone optimally) is decidedly less clear and demands a much deeper and nuanced analysis of how systems comprising this subsystem and how other subsystems, data, and tools will ultimately be packaged, tested, deployed, and consumed.

In the following, we concisely form the foundational criteria needed for determining whether a given assumption is — or may reasonably be anticipated to become — one whose truth can be deduced purely from information available in some well-defined physically proximate region of the system. If so, then checking that assumption at run time is — or is anticipated to become — entirely redundant and amenable to a (optional) defensive check. Otherwise, the assumption remains one of validating intrinsically external input, thereby requiring the check to be present uniformly and unconditionally, i.e., in *every* build mode.

Neighborhoods

As previously suggested, defensive checks are aimed at confirming the truth of assumptions that must always be true in a properly implemented system. More specifically, a check is *defensive in nature* if it is — or is anticipated to become — one whose truth can *always*[8] be proven from information that is proximately available within some well-defined *physical*[9] region, which we will refer to generally as a

[8] By "always," we mean that there is *never* any expectation that this particular check will *ever* be deployed in a way where the correctness of overall operation depends upon that check being performed.

[9] We use the term "physical" here to connote that which is collocated in a manifestly inseparable material way (e.g., a source file, an executable) beyond mere *logical cohesion* (e.g., `namespace`).

neighborhood.[10] An *immediate neighborhood* is an atomically cohesive physical region — e.g., a source file — that is devoid of any pertinent conditional compilation or source-file inclusion that might render otherwise provably valid assumptions suspect.

Purely Defensive and Contextually Defensive Checks

A specific check is *purely defensive* if the author or reviewer of the check can (at least in principle) prove — and perhaps be wrong (see Figure 2) — that this check must *always* pass given just the information that is available in the immediate neighborhood of the check. In other words, regardless of the circumstances of how the code containing the check is used and deployed,[11] a purely defensive check cannot (in theory) be violated (but occasionally is in practice).

Often, however, a check that is intended to be defensive and entirely optional (e.g., a precondition check) will not have sufficient information bound into its immediate physical neighborhood to prove (or otherwise ensure) that — irrespective of how the subsystem in which it is embedded is used — the assumption being checked cannot be violated. Such a check, though defensive in nature, does not rise to the level of being purely defensive. When specificity is needed, we will refer to a nonpure defensive check as being *contextually defensive*.

Contextually defensive checks are intentionally redundant and inherently optional runtime checks that are anticipated to be used only as a part of a larger system that will be packaged, tested, deployed, and consumed (PTDCed) as an indivisible physically cohesive unit embodying sufficient information to prove that the assumption being tested by the check is necessarily true, irrespective of whether that check is performed. The most obvious and common form of a contextually defensive check is one that validates a function precondition — e.g., `assert(value >= 0)` — of a function (e.g., `sqrt`) such as might be defined in a (reusable) library (e.g., `std`).

---

[10] Note that our definition of a neighbourhood 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 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]**.)

[11] When assessing what constitutes a purely defensive check, we entertain only *reasonable* (i.e., responsible, productive, nonmalicious) coding practices. For example, using the preprocessor to somehow change the meaning of identifiers or modifying the assembly output (post compilation) would violate our premise of reasonable practice.

Internal Assertions and Postconditions

Whether we consider a postcondition to be *contextually* versus *purely* 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 assertions that depend on preconditions being satisfied. Again, to consider a check purely 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 purely 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 assertions 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 *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.

Preconditions

A precondition is both a requirement imposed on each caller of a function and an assumption that the implementer of the function may presume to be true. Libraries (especially reusable ones) have historically presented points of contention with their implementers trusting (let alone *assuming*, i.e., for optimization purposes[12]) that a given precondition is always met. Although developers seem to widely accept that violating a precondition check is incontrovertibly a software defect and that

[12] In some (proposed) defensive-checking facilities, if a check is not actively performed, then the compiler is permitted to presume that the assumption is unconditionally true and to optimize accordingly. Whether such a feature is desirable or useful (as of February 2020) is a matter of active research and debate within SG21 (the Contracts Study Group) of the C++ Standards Committee.

attempting to check such assumptions is inherently defensive in nature, they can harbor a strong reluctance to ever disabling such checks for fear that, someday, such an assumption might suddenly be violated. Though the users of such a library are not typically known to a library author, the library functionality is (or should be) designed specifically to be PTDCed as an inseparable part of a larger, cohesive software system wherein the programmatic clients of the library can reasonably be expected to uphold and enforce the (presumably thoroughly documented[13]) library requirements. This presumption of PTDCed entities is precisely what justifies the classification of precondition checks as being (contextually) defensive in nature and therefore implemented as independently and externally configurable rather than hard coded or tied to a function's parameters and/or essential behavior (as specified in its contract).

Although libraries may be (re)used by many (e.g., application) clients, each client of a library uses that library in its own specific way. As soon as a component embodying a contextually defensive check has been inseparably bound into a physically cohesive subsystem (a.k.a. neighborhood) containing sufficient information to prove or otherwise know that the assumption being checked is necessarily true in any context in which the composite (i.e., client) subsystem might reasonably be used, the contextually defensive check (in theory) no longer serves any practical purpose. With respect to this specific cohesive subsystem, the heretofore contextually defensive check is now *manifestly defensive* and can (at least in theory) now (or, in practice, at some point) be safely disabled.

Defensive Checks in Larger Systems

Contextually defensive checks are not necessarily limited to a single executable and might well deal with larger, more inclusive systems involving other programs, data, tools, and so on as long as the PTDC criteria are eventually satisfied. For example, a configuration file read at run time might be considered to provide consistently and permanently reliable information if the executable along with the configuration file are intended to be PTDCed together (with no comprised parts ever being modified subsequently) as, say, a container image. Furthermore, even network communication among multiple services deployed in a cluster might be deemed trusted by the engineering teams developing these distinct services; hence, checks verifying their validity could be considered defensive in nature.

Confirming that information external to an executable will be received reliably (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, all the physical hardware must be sequestered within a physically confined and secured area (e.g., a proprietary data center). If such effort is not justified

[13] In addition to proper testing and deployment strategies, ensuring proper communication among the (possibly many, distinct) developers of the subsystems is also important. Providing contract descriptors in a natural language greatly facilitates such communication. See **khlebnikov20b** for an in-depth (mostly programming-language-agnostic) analysis of how such contract descriptions are presented effectively and synergistically with typical defensive-checking facilities.

— or perhaps even impossible, e.g., due to the services being accessible to anyone on the Internet — then treating the communication among even concurrently deployed subsystems as satisfying even the spirit of this PTDC criteria is flat-out wrong, irrespective of the precise mode of communication (e.g., sockets, named pipes, shared memory segments, direct calls to a runtime-loaded shared library, or via language bindings).

Summary

Use of defensive-checking frameworks is reserved for checks that can be reasonably classified as being either purely or contextually defensive. Checks that are not expected to eventually satisfy the PTDC criteria are not defensive in nature and are therefore ill suited to such frameworks. Whether a check can be reasonably considered defensive in nature is usually obvious, sometimes subtle, and, on rare occasion, debatable (see the following section). The definitions provided above are reprised concisely for the reader's convenience in Figure 6.

> **Contextually Defensive Check** — A defensive check that is not purely defensive, i.e., one whose truth cannot be proven from its immediate neighborhood yet is defensive in nature; hence, it is anticipated that, in every case where the check is part of an entity that is consumed by external users, sufficient information will 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., purely defensive) or else 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 (or perhaps cannot reliably) satisfy the PTDC criteria for the entity.
>
> **Immediate Neighborhood** — The physically contiguous (monolithic) region surrounding the implementation of a defensive check, devoid of constructs that might reasonably cast doubt as to whether the otherwise noncontextually defensive check is in fact manifestly defensive (e.g., conditional compilation and `#include` directives between the check and the information required to prove the truth of the checked assumption).
>
> **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 be incorporated for consumption by external users.
>
> **Neighborhood** — A physical subregion of an entity containing a defensive check that when PTDCed would be sufficient to render that check manifestly defensive.
>
> **Purely Defensive Check** — A manifestly noncontextually defensive check, i.e., one whose unconditional redundancy (i.e., within *every* syntactically correct program) can be proven locally (e.g., by a human reviewer), irrespective of whether and how its local neighborhood is ultimately bound into other entities for consumption by external users.

PTDC —*Package, Test, Deploy*, and *Consume*

PTDCed —*Packaged, Tested, Deployed*, and *Consumed.*

PTDC Criteria — A criteria applied to an entity that will be made available for consumption by external users wherein any of its initially physically separable constituent parts have been PTDCed together, yielding one immutable physically cohesive (atomic) unit comprising them all.

UB-safe — The property of a (e.g., defensive) check that indicates the check itself cannot possibly trigger undefined behavior in any build mode, e.g., because any undefined behavior that might have been introduced by its predicate is guarded by either (permanent) validating code or some other defensive check that would necessarily be active and occur earlier in any conceivable build mode in which the original check was active.

Shadowed — The property of a defensive check that indicates the check itself cannot possibly trigger *new* undefined behavior in any build mode because, for the undefined behavior that might be introduced by its predicate, the identical form of undefined behavior is also introduced in either (permanent) validating code or in some other defensive check that would necessarily be active and occur (either earlier or later) in any conceivable build mode in which the original check was active.

Figure 6: Summary of terms pertaining to defensive checks.

Real-World Assumption-Checking Scenarios

Armed with a thorough understanding of what conceptually distinguishes defensive checking from input validation, we now present a sequence of real-world examples that cross the narrow divide separating the two.

Internal Logic Checks

Checks that verify essential properties of implemented algorithms naturally satisfy the requirements of defensive-in-nature checks in that, in any defect-free program, they are (by definition) redundant. 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 (see Figure 2) more constrained — algorithm arrives at the same result. Figure 7 illustrates all three of these sorts of checks, which, in this instance and considering that the checks can be shown to hold true using information derived *exclusively* from their immediate neighborhood, can each be accurately classified as being purely defensive.

```
bool containsSamples(const std::vector<int>& rawData,
                     const std::vector<int>& transformedSamples)
{
    std::vector<int> data{rawData.size()};
    std::transform(rawData.begin(), rawData.end(), data.begin() &transformDatum);

    // Sort the two halves and either merge them or discard the second half.
    auto mid = data.begin() + data.size() / 2;
    std::sort(data.begin(), mid);
    std::sort(mid, data.end());
```

```
    if (useBothHalves(data.begin(), mid, data.end())) {
        std::inplace_merge(data.begin(), mid, data.end());
    } else {
        data.resize(std::distance(data.begin(), mid));
    }

    // About to start binary searching - 'data' will be sorted regardless of
    // which branch was taken above.
    assert(std::is_sorted(data.begin(), data.end()));

    for (int sample: transformedSamples) {
        auto first = data.begin();
        auto last  = data.end();
        auto count = data.size();

        while (count > 0) {
            auto step = count / 2;
            auto it = first + step;
            if (*it < sample) {
                first = ++it;
                count -= step + 1;
            }
            else {
                count = step;
            }
        }
        // 'count' will be exactly 0 after the loop.
        assert(0 == count);

        bool found = first != last && *first == sample;
        if (!found) {
            // A linear search always arrives at the same result.
            assert(data.end() == std::find(data.begin(), data.end(), sample));
            return false;
        }
    }

    return true;
}
```

Figure 7: Examples of purely defensive checks.

Unreliable Input Sources

If a subsystem has as sources of input entities that are not within the control of that subsystem, no physical neighborhood that encompasses both the input and checks validating the input can reasonably be defined. The PTDC criteria cannot apply to such checks; therefore, for the designers of that subsystem to classify them as defensive would be irresponsible. Hence, presuming that any such externally supplied nonprogrammatic input is potentially flawed and being prepared to handle such flawed input accordingly is always wise. Even if the desired course of action for malformed input is to abort the program on failure, this should not be performed with a defensive check so that the verification will be applied in every build mode.

For example, failing to consistently validate input that might be received from even a well-intentioned human operator will inevitably lead to unpredictable intermittent failures. As a second example, input received by a public HTTP server might not be simply accidentally malformed but could also arrive from a malicious actor aiming to destabilize the system; all such input requests should therefore always be validated thoroughly. Figure 8 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 8: An example of a (poorly engineered) public-facing HTTP server that misuses a defensive-checking framework (namely `<cassert>`) to perform input validation.

Precondition and Postcondition Checks

A function's contract may impose certain *preconditions* — i.e., semantic limitations on syntactically valid inputs and/or ambient 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 essential (or any other) behavior is affected. 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.

Although the client invoking a library function (including one having preconditions) is generally unknown to the function, the caller and callee are nonetheless expected to eventually become part of a larger logically cohesive entity that is PTDCed as an inseparable physical unit. Therefore, classifying precondition checks as contextually defensive and employing a defensive-checking framework to detect inadvertent function misuse by its (trusted) programmatic clients is a reasonable practice.[14] For

[14] 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

example, a `binarySearch` function extracted from the `containsSamples` function in Figure 7 might require, as a precondition, that the input range be sorted, as illustrated in Figure 9.

```
// 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 <= *it);

        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;
    // Postcondition check (3).
    assert(result == (last != std::find(first, last, value)));
    return result;
}
```

Figure 9: Binary search function that uses defensive checks for its precondition checks (1) and (2) as well as its postcondition check (3).

Note that in contrast to Figure 7 (where a binary search was performed in the context of another, larger function), after factoring out independently callable `binarySearch` function, the `is_sorted` check, while still a defensive in nature, changed its category from a purely 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. This duality reflects both the intuition behind why precondition checks are defensive in nature and also how a change in the physical

---

only is such a check explicitly *not* part of its contract, but in some cases doing so might be prohibitively expensive if not impossible. By employing a defensive-checking framework, such as BSLS_ASSERT or the one originally proposed for C++20, that affords the ability to enable inexpensive checks (e.g., *default* level) without necessarily enabling more expensive ones (e.g., *audit* level), a library developer can provide a diverse set of clients with better control over apportioning runtime resources commensurate with their own respective states in the software development lifecycle.

neighborhood of the check might well affect its classification (i.e., purely versus contextually defensive).

In a similar manner, postconditions also correspond to purely defensive internal logic checks and yet hyper-technically cannot be classified as purely defensive if they depend on the degree of trueness of any of the function's preconditions. In practice, however, most contextually defensive postcondition checks of robustly written libraries are guarded by the precondition checks enabled or disabled together with the postcondition checks, implying that the postcondition checks would not typically be reached should preconditions be violated.

Moreover, violating a precondition, which is considered soft (library) undefined behavior, can easily lead to hard (language) undefined behavior by running afoul of the assumptions implicit in the function's implementation itself. Hence, even if classified as input validation, the postcondition check would be of little use if the function's preconditions do not hold. To facilitate local reasoning, a common practice is to classify (misclassify) postconditions (and, similarly, internal logic checks) as being purely defensive even when they presume that the preconditions are true.

Furthermore, when invoking an external function, the standard practice is to assume that the function is implemented correctly (and tested thoroughly); hence, postconditions can, for all practical purposes, be considered guaranteed to be correct with respect to any local proof of correctness that makes use of them. Without such an assumption, any hope for the scalability of such local correctness proofs would be lost.

Sidebar: Precondition Checks in Hierarchically Reusable Libraries

During the development of hierarchically reusable software,[15] it is not uncommon for a piece of low-level functionality to be used locally in other functionality where its preconditions checks are initially purely (and hence manifestly) defensive, and then later, after fine-grained physical factoring, only contextually defensive, as evidenced in Figures Figure 7 and Figure 9, 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, thereby qualifying a check for this assumption to be considered (contextually) defensive (often with no need to return status); yet other clients 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

---

[15] A *hierarchically reusable library* is a library 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.

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] While 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), providing two entirely distinct functions — each customized to suit its respective client's manifestly different needs — is almost always wise. Figure 10 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]
        // 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 10: Example rendering of a dual API (nonvalidating alongside validating).

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 11.

```
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; }
```

[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.

[17] When following the coding conventions used in the BDE family of libraries, the name of the function with the validating API would instead affix the suffix `IfValid` with the resulting identifier being `parseIfValid`.

```
    std::vector<DateTimeInterval> intervals;
    std::string line;
    while (config >> line) {
        DateTimeInterval interval;
        if (0 != DateTimeIntervalUtil::tryParse(&interval, line)) {
            std::cerr << "Invalid date-time interval encountered.";
            return 3;
        }
        intervals.push_back(interval);
    }

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

Figure 11: Properly validated (user-supplied) configuration.

If, however, the developer intended these resources to be modified by only the application developer, we might consider whether some (or all) of these checks could be considered defensive in nature and simply assert them. This idea is enticing for the potentially computationally intensive and repeated checks performed by `tryParse` in the tight loop, affecting the application startup time. However, discounting the possibility of human beings making a mistake when defining or editing these resources would itself be an error.

Even if systemwide testing confirming the validity of the external resources is performed postdeployment, relying on this validity to continue even when any part of a PTDCed system can be modified after deployment (without being re-PTDCed) violates the PTDC criteria imposing no postdeployment changes to the constituent parts (and is asking for trouble). Expecting text quickly typed into a command line by a human being to be valid is more dubious still. For such checks to be justifiably classified as manifestly defensive, the PDTC criteria must be satisfied.

Since the engineer, no matter how qualified and careful, is not part of a physically inseparable unit that undergoes packaging, testing, deployment, and use along with the rest of software system, any changes made by such actors will necessarily lead to the entire system being re-PTDCed. In the case of a command line, the input will need to be captured statically, say, within a script. With a fully encapsulating process in place (e.g., one that uses containerization), reclassifying the checks as being defensive in nature may be feasible, and these checks can then eventually be disabled in production once sufficient hardening has occurred, thereby affording better startup performance. Figure 12 illustrates how such containerized application might be implemented.

```
int main(int argc, const char *argv[])
{
    assert(2 == argc);

    std::ifstream config(argv[1]);
    assert(config);
```

```
    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);

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

Figure 12: Containerized application with configuration checks reclassified as defensive.

When it comes to internal data files, other simpler alternatives to containerization fully satisfy 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()` or `std::terminate()`. In this way, we can effectively use the containerizing properties of the program's executable image cheaply and effectively to ensure that the external files do not change independently of the overall system.

Larger Neighborhoods

Thus far we have 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.

First, imagine that we have a request/response system running on a single, very large, multicore supercomputer that handles concurrent users by spawning numerous identical processes. Imagine further that the state of each active client session can be maintained 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,[18] we might choose to enable those checks

[18] 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 — except, perhaps,

when designing the infrastructure to spin up the system. Once that mechanism is sufficiently proven, we may eventually choose to disable those invariant checks while making no other changes to the system. In a defect-free system, we can know that all object invariants will hold irrespective of whether they are checked (redundantly) at run time; hence, this single-computer multiprocessing scenario satisfied the PTDC criteria.

Next, let's imagine that we have 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 neighborhood, provided, of course, that all the constituent parts can be reliably PTDCed and then secured such that no part is subject to independent modification.

As our final example, let's now imagine a large data center, such as might be found at a major financial information services company. These massive computing facilities, containing an untold number of server machines, must run continuously 24/7, with no illusion that they can ever all be stopped, updated, tested, and redeployed in unison, and yet — at this scale — eliminating even a few cycles per customer request can translate to significant savings in terms of reduced hardware footprint, heat dissipation, and so on. So how can local defensive checks possibly help here?

For illustration purposes, imagine we have a (small) computing center consisting of just 100 machines (arranged in a ten by ten array, **`M[10][10]`**) each running (on average) roughly 1000 (nearly) identical processes (10^5 processes in total) that perform (essentially) the same request/response functionality. 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 (in production) by live customers.

Unit tested or not, deploying new software to production is a delicate task, and, of necessity, we must proceed carefully. Hence, a new version of our server process is never rolled out to our user machines all at once but in increasing (e.g., quadratically) *waves*. First, we bring down a single user (server) machine, say **`M[0][0]`**, replace the current version of the executable with the new one, and then proceed to spin up all the processes on that machine. Then we bring it online and basically wait to see what happens. If, after some time, no problems arise, we continue the rollout by bringing

---

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).

up, say, three more machines, e.g., **M[1][0]**, **M[1][1]**, and **M[0][1]**, and again we wait. With each successive wave, we roll out more machines until all of them are executing processes spawned by the same new executable. If at any point we discover a problem, then we reverse the process and return to our 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, our (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 we proceed differently? We might start by deploying the slower but more robust defensively checked version on **M[0][0]** and see what happens there first. With defensive checking enabled throughout every process on that first machine, any violations of defensively checked internal assumptions (i.e., those relating specifically to the correctness of the process itself) will be quickly flagged, and we can abort the next wave much earlier. If, after a short delay, no such problem is reported, we can then proceed (much more quickly than without defensive checking enabled) with the second wave and so on.

Since defensive checking requires additional computer resources, we will not want all of our hardware to be doing such redundant runtime checking for long. Even as the first wave continues to spread over our computer farm, we can begin to introduce a second wave that replaces the defensively built executables with leaner, nondefensive ones — now with greatly reduced concern that these new, higher performance executables will be disruptive in production.

What makes this approach fundamentally sound is the understanding that (1) the executable itself is PTDCed and then tested as a nonmodifiable unit before it is ever deployed to production, and (2) once all of the new executables are deployed and running in unison, they too have effectively been PTDCed as a yet larger nonmodifiable unit — i.e., the entire computer farm can, in effect, be considered one gigantic neighborhood! Hence, once everything appears to be working well, we need not endure 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).

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, we might choose to leave a few machines running the slower, more robust, defensively checked version in place. In this way, we can titrate the cost of performing statistically significant, practically useful defensive checking 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 arbitrary size. Instead of having just a two-dimensional 10 x 10 grid, imagine

a three-dimensional block[19] of machines `M[100][100][100]` in an edifice the size of a warehouse (10^9 processes). The same sort of two-phased, multiwave rollout approach (while keeping just a tiny fraction of the machines enabled for statistically useful defensive checking) pertains. A concise summary of the various principles elucidated in this and the preceding subsections for selecting defensive checking versus input validation in real-world scenarios are summarized in Figure 13.

Subsection Title/Topic	**Principles Being Demonstrated**
Internal logic checks	Immediate physical neighborhood Purely defensive checks
Unreliable input sources	External users Input validation
Precondition and postcondition checks	Contextually defensive checks PTDC Manifestly defensive checks Shadowing other sources of undefined behavior
Resource files	Evolving physical neighborhood
Larger neighborhoods	PTDC achieved through replication Staged rollout with defensive checking Statistically significant partial checking

Figure 13: Brief summary of principles elucidated per subsection.

CONCLUSION

Making assumptions is inherent to writing any software system. We have identified two distinct and nonoverlapping kinds of assumptions: (1) those pertaining to the correctness of a software system itself (i.e., the system does what it is expected to do) and (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 (e.g., massively replicated) 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).

[19] Note that heat dissipation can become a governing factor, especially in a three-dimensional block.

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 gross 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).

Even if the assumptions are properly categorized, it is important to be mindful of potential failures associated with side effects and potential introduction of (language) undefined behavior in predicates of defensive checks. In particular, in defensively checked predicates, side effects that affect essential behavior are defects and those that don't affect behavior might be considered *benign* or at least *tolerable*. With respect to introducing undefined behavior, any defensive check may be *UB-safe* (no additional undefined behavior possible) or at least *shadowed* (by the same undefined behavior) in every build mode. Understanding these distinctions underlies proper practical implementation of defensive checks, including reasonable predication of postcondition and other internal checks on the degree of trueness of all preconditions.

We have identified several important properties related to successfully characterizing whether a given assumption will ultimately be knowable before run time or else (for the system to be defect-free) always require runtime validation (see Figure 6 for a comprehensive taxonomy). A check is (1) *defensive in nature* if its degree of trueness is — or is anticipated always to be — deducible from information proximately available in some well-defined physically cohesive region called a *neighborhood*; (2) *purely defensive* if it can be proven — irrespective of the context in which its (intrinsically physically contiguous) *immediate neighborhood* resides; and (3) *contextually defensive* if it satisfies the first of the above definitions but not the second.

A system comprised of smaller, physically separable entities that are *PTDCed* together forms a neighborhood for a (contextually) defensive-in-nature check if the degree of trueness of that check can (in principle) be proven when the system is packaged, in which case the check becomes *manifestly defensive* with respect to this specific deployment.

As the seminal contribution of this paper, we nominate the *PTDC criteria* as the measure by which to adjudicate whether a check can be considered defensive in nature (i.e., eventually provably redundant) and, hence, removed without affecting either the correctness or essential behavior of a defect-free program. The PTDC criteria require that a PTDCed system — i.e., one having a neighborhood sufficient to prove a given assumption — is unilaterally controlled such that none of its constituent pieces is susceptible to postpackaging modification.

When applied to a series of increasingly nuanced real-world examples (see Figure 13), the PTDC criteria quickly and clearly exposed the fundamental nature of several classically difficult-to-categorize assumptions. Defensive checks (for security reasons as well as correctness) are never appropriate for assumptions that involve knowledge that emanates or propagates through any part of the overall system (e.g., via a public

network) not under the complete authority and control of the overall system owner. Furthermore, they are not amenable to even trusted direct (raw) human input (e.g., command line, console, control file) unless such input is captured (e.g., in a script or data file) and PTDCed along with the system such that no part is subsequently modifiable. And yet certain assumptions that span processes and even machines can — with sufficient diligence — still be treated as part of a very large neighborhood and therefore amenable for defensive checking (e.g., when multiple instances of essentially the same process are running as part of a larger multiprocessing system). Such diligence was motivated in that, for a massively parallel multiprocessing system, statistically valid and very useful information can be collected — at substantially reduced runtime overhead — simply by enabling defensive checking in only a small fraction of the otherwise identical production processes.

Finally, we posit 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); hence, 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, we must always be mindful that some assumptions are inherently defensive in nature while others are not.

The goal of this paper was and is to elucidate — to all developers — how to correctly discriminate between two important and disjoint assumption categories and, hence, when the use of a defensive-checking framework, such as `<cassert>`, is appropriate.[20]

REFERENCES

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

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).
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.

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

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