Core safety Profiles: Specification, adoptability, and impact

Document Number: **P3081 R0**
Date: 2024-10-12
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG15, SG23, EWG

Contents

1 Overview: Motivation in a nutshell 2
2 Approach: Strategy and tactics 3
3 **`type_safety`** Profile 8
4 **`bounds_safety`** Profile 12
5 **`initialization_safety`** Profile 14
6 **`lifetime_safety`** Profile 15
7 **`arithmetic_safety`** Profile 17
8 Adoptability and impact: Broadly applicable strict mode 19
9 Addition: Null-terminated `zstring_view` 20
10 References 21

Abstract

This is one of two companion papers.

(1) My "C++ safety, in context" paper, published as a blog essay, contains the full motivation, context, and rationale -- please see that paper for those topics, written for a broad audience.

(2) This paper contains the concrete proposed semantics, written for WG21 experts:

- Supports [P3038R0] to standardize an initial set of urgently needed enforced safety Profiles, in addition to any other Profiles.
- Described how a Profiles implementation can prioritize **adoptability** and safety improvement **impact**, especially to silently fix serious bugs in existing code by just recompiling the code (i.e., no code changes required) where possible, and making it easy to opt into broadly applicable safety profiles.
- Suggests we could consider adding new kinds of normative requirements on C++ implementations, specifically to:
    - offer selected specified reliable and automatable source code modernizations, as part of the C++ implementation rather than in a separate or third-party tool;
    - improve `dynamic_cast` itself (or, to preserve ABI, provide a broadly adoptable alternative), by requiring an implementation support it in all modes with a run-time performance guarantee.

1 Overview: Motivation in a nutshell

1.1 Motivation

Our problem "is" the ease of writing code in C++ that inadvertently creates security vulnerabilities due to weak language safety guarantees, very often due to the lack of enforcement of practical, established best practices.

The most urgent need is to address type, bounds, initialization, and lifetime safety; initially targeting those four areas can significantly reduce security vulnerabilities (CVEs), after which we should also address further areas.

Our problem "isn't" figuring out which are the most urgent safety issues; needing formal provable language safety; or needing to convert all C++ code to memory-safe languages (MSLs).

1.2 Constraints

We must maintain backward compatibility and not require changes to C++'s object model and lifetime model.

Each enforced rule must be deterministically decidable at compile time in a way that is sufficiently efficient to implement in-the-box in the C++ implementation (including without unacceptable impact on compilation times).

1.3 Standardized Profiles

This paper follows SG23's current direction per [P2816R0] of pursuing enforceable safety Profiles for C++.

This paper supports [P3038R0]'s call to specify roughly the same set of enforceable Profiles and features in the Standard. These are the urgently needed Profiles (and I'm happy with any naming/granularity that covers these):

This paper's name	**[P3038R0] name and section**
`type_safety` (casts, unions, varargs, etc.)	`type_safety`, throughout the paper
`bounds_safety` Ban pointer subscripting Global range checking	`ranges`, §10-11 Ban pointer subscripting Global range checking
`initialization_safety` Require initialization, explicit opt-out	`type_safety`, §5 Require initialization, explicit opt-out
`lifetime_safety` Global null dereference checking Lifetime static analysis Ownership Dangling Invalidation	Pointers and owners Global null dereference checking, §6 Lifetime static analysis Ownership, §7 Dangling, §8 Invalidation, §9
`arithmetic_safety` (narrowing, signedness, overflow)	`arithmetic` (narrowing, signedness, overflow), §12

To reduce risk and maximize the chance of consensus, all initial Profiles should target well-known urgent needs, and be directly based on rules from known implemented prior art. All of the rules in this paper have been implemented, and most have been used in production code.

2 Approach: Strategy and tactics

2.1 Strategy (per current SG23 direction and Stroustrup P3038)

Define standard enforced "Profiles" that a conforming C++ implementation must enforce when enabled, notably `type_safety`, `bounds_safety`, `initialization_safety`, `lifetime_safety`. This is in addition to any user-defined Profiles.

Each Profile consists of rules. Each rule must be deterministically decidable at compile time (even if it results in injecting a check enforced at run time), and must be sufficiently efficient to implement in-the-box in the C++ compiler without unacceptable impact on compile time.

Rules are portable and enforced in the C++ implementation, not in a separate tool such as a static analyzer. Note this good summary by David Chisnall in a January 2024 FreeBSD mailing list post, [Chisnall2024]:

> *"Between modern C++ with static analysers and Rust,* ***there was a small safety delta****. The recommendation* [to prefer Rust for new projects] *was primarily based on a human-factors decision:* ***it's far easier to prevent people from committing code that doesn't compile*** *than it is to prevent them from committing code that raises static analysis warnings. If a project isn't doing pre-merge static analysis, it's basically impossible."*

Opt-out is explicit. When a source file is compiled in `[[enforce(P)]]` mode for a Profile `P`, use `[[suppress(P)]]` or `[[suppress(P, "justification message")]]` (as proposed in P3038) to say "trust me" and opt out of Profile `P` in that scope.

Opt-out is granular by scope. We should probably allow writing suppression on at least a statement, a block scope, a variable or function definition, a class, a source file, a translation unit (e.g., command line switch), and a project (e.g., IDE setting), all of which have prior art (see Note below). For example, when this code is compiled in "`type_safe`" mode, it has the commented compile-time meaning, because the cast is disallowed in the `type_safety` Profile:

```
void f(int i)
{
    (double*)&i;  // error: type-unsafe
    [[suppress(type_safety)]]
    (double*)&i;  // ok
}
```

```
[[suppress(type_safety)]]
void f(int i)
{
    (double*)&i;  // ok
}
```

```
void f(int i)
{
    (double*)&i;  // error: type-unsafe
    [[suppress(type_safety)]] {
        (double*)&i;  // ok
    }
}
```

```
// compiled without [[type_safety]] enabled
void f(int i)
{
    (double*)&i;  // ok
}
```

Note These specific granularities to opt-in/out can be further adjusted, but are chosen to follow existing practice: MSLs like Rust and C# commonly support `unsafe { }` blocks, functions, and classes/traits. Existing command-line/project level enforcement flags commonly used today to opt in/out of warnings including existing safety-related warnings.

Opt-out is granular per Profile. When opting out of one specific Profile, other Profiles are still enforced. This prevents explicitly opting into one "trust me" accidentally also disabling unrelated checks. For example, when this code is compiled in "`type_safe` and `bounds_safe`" mode, it has the commented compile-time meaning, because the cast is disallowed in the `type_safety` Profile and the pointer arithmetic is disallowed in the `lifetime_safety` Profile:

```
void g(void* pv)
{
    ++(double*)&i;  // error: type+bounds
    [[suppress(type_safety)]]
    ++(double*)&i;  // error: bounds
    [[suppress(type_safety)]]
    [[suppress(bounds_safety)]]
    ++(double*)&i;  // ok
}
```

```
[[suppress(type_safety)]]
void g(void* pv)
{
    ++(double*)&i;  // error: bounds
    [[suppress(bounds_safety)]]
    ++(double*)&i;  // ok
}
```

2.2 Tactics (for discussion and feedback)

We can address each rule violation using any of three basic tactics, with the noted suggested order:

	Tactic	Adoptability / UX	Manual code changes?
Fix (ideal)	**If efficient and feasible**, give the code the intended and safe semantics (i.e., make the code do the right and safe thing) "Efficient" can include providing guarantees (or evidence such as binary object comparison) of functionality and performance Such a semantic improvement should always apply to both "enforced" and "non-enforced" mode, so that code that compiles cleanly in both modes means the same thing	"Holy grail": Automatically fixes bugs in existing code just by recompiling the code Plus some cases can perform reliable automatic source M modernization	None ✓
Reject (great)	**Otherwise**, diagnose violations at compile time, and where possible provide a clear "use this instead" correction This is always "Ill-Formed, Diagnostic Required"	"If it compiles then it's safe" Plus some cases can perform reliable automatic source M modernization	Violations must be addressed by manual or automatic code changes

Check (good)	**Otherwise**, diagnose violations at run time, and let the program customize how violations are handled (ideally the same as contract violation handlers)	Automatically diagnoses bugs in existing code just by recompiling the code	None ✓

The suggested order prioritizes (1) the zero-overhead principle (as efficient as writing equivalent code by hand, and a program pays the overhead only if it uses the feature), then (2) zero source code changes wherever possible. Those two, in that order, should take priority over all other considerations, including even over code clarity.

I expect priority (1) needs no debate. We accept the zero-overhead principle as table stakes, otherwise we're not C++ anymore.

However, (2) might be surprising, and we should explicitly discuss and poll it.

"New/updated" code vs. "existing code": Customer motivation and demand. As we address the safety problem we have many solutions for "new/updated" code, ranging from "rewrite in Rust" to "use a subset of the language" (e.g., via Profiles). Customer feedback I've received includes that even the latter can require rewriting code; as good and simple a rule as "use `std::span`" in practice can require non-local code reorganization, and this is known to impede adoption. But we have seen very few solutions proposed for "existing code." And customers have asked the legitimate question: "Well, what about all the existing C++ code I can't afford to manually migrate?"

We must minimize the need to change existing code. For adoption in existing code, decades of experience has consistently shown that most customers with large code bases cannot and will not change even 1% of their lines of code in order to satisfy strictness rules, not even for safety reasons unless regulatory requirements compel them to do so. Reasons include, but are not limited to: the work could require many person-years which can be impossible regardless of the benefit; changing code even to fix bugs always creates new bugs at some rate so change is never free; and some code is impossible to change because it is not owned or is no longer understood.

> **Example:** I suggest making `static_cast<D*>(pb)` pointer downcasts, where `pb` is a pointer to a polymorphic base type `B` and where `D` is derived from `B`, be a "Fix" rather than a "Reject," as follows:
>
> - We should **normatively require** that the compiler actually perform a safe `dynamic_cast`, without changing the code which still says `static_cast`.
> - We should **normatively require** a C++ implementation to offer to automatically change the source code itself to say `dynamic_cast`. (Note: "C++ implementation" is intentional, not a separate tool. It can be as simple as emitting a diff or patch file and shelling out to `git apply xxx.diff` or similar.)

This intentionally suggests favoring ease of adoptability (no code change required, just recompile the code and the bug is fixed) at the potential expense of arguably making the code misleading if the offered automated rewrite is not accepted (if the source code is not changed it will still read as "static cast," and users familiar with pre-profiles static casts who have not yet become familiar with safety profiles may assume the code is unsafe). Because incorrect `static_cast` downcasts are a persistent source of real vulnerabilities, I suggest we should favor fixing as many as possible (i.e., those involving polymorphic types) with just a recompile, and after that fixing as many more as possible by offering a reliable automatic source code rewrite to fix them (e.g., if the type is non-polymorphic, offer an automatic source code rewrite to make the base class destructor virtual).

We must minimize impact even on new/updated code, in particular by not relying on frequent annotations. Even for new/updated code (when programmers are manually writing code in the new "enforce SomeProfile" mode, including both new code and when changing code during maintenance), any requirement to annotate a significant amount of code is a serious adoption impediment. Many efforts (e.g., Cyclone, CCured), have foundered on this rock and have been ignored in the marketplace; only rare exceptions exist where heavy-annotation approaches have been adopted at scale, and those are limited to narrow success in a specific company that mandated them and made a massive investment (e.g., SAL).

> **Suggested limit metric:** Any annotation requirement we create even for new/updated code must not exceed the usage frequency of "unsafe" keywords in C#, Rust, and similar languages. The easiest way to achieve that is to principally use only annotations having such a meaning (i.e., to opt out of safety in a particular setting, as this paper suggests).

These tactics can allow us to enable two "levels" of adoption, where the lower level maximizes initial adoptability and impact, and the higher level can require code changes and delivers full impact:

Adoption level for a given Profile P	**What parts of P are enabled**	**Adoptability**	**Impact**
`suppress(P)`	None	n/a — ordinary no-guardrails C++	None
`apply(P)`	Fix ✓ Check ✓ Modernize ✓ Optionally, emit Reject as warnings (optional so as not to break warnings-as-errors)	**Zero manual source code changes** All existing code still compiles, "It Just Works" Plus offer reliable automatic "fixit" improvements	**Some safety benefits "for free [in terms of manual code changes]": all automatic fixes and checks applied**, all other improvements are non-mandatory suggestions **"Just recompile with Profiles enabled and start seeing benefit"**
`enforce(P)`	Plus Reject ✓	Plus Reject violations will break the build, and require code changes	Full safety benefits

2.3 Profiles and rule overview: "1-pager" cheat sheet

The following table is an overview summary of the rest of this section. As described in the previous section, the priorities are:

- Fix statically where efficiently possible, otherwise Reject statically where possible, otherwise Check dynamically.
- Modernize code automatically wherever possible.

Note In this table, "Check" means adding additional dynamic checking, beyond existing dynamic checks.

Profile	Rule	Tactics: Fix/Reject/Check F R C
`type`	`reinterpret_cast` in all cases (Type.1.1)	– R –
`type`	`const_cast` in all cases (Type.3)	F M R –
`type`	`static_cast` problematic cases (Type.1.2, Type.1.3, Type.1.4, Type.2)	F M R C
`type`	`dynamic_cast` redundancy and performance (Type.1.3, Type.1.4)	F M – –
`type`	`(c_style)cast` problematic cases (Type.4)	F M R C
`type`	`functional_style(cast)` problematic cases (Type.4)	F M R C
`type`	`union` in all cases (Type.7)	F – C
`type`	`va_arg` in all cases (Type.8)	– R M –
`bounds`	Pointer arithmetic in all cases (Bounds.1, Bounds.3)	– R –
`bounds`	Array-to-pointer decay in all cases (Bounds.3)	– R –
`bounds`	Subscript checking including arrays/`vector`/`span`/etc. (Bounds.4)	– – C
`initialization`	Member variables (Type.6)	– R –
`initialization`	Non-member variables (Type.5)	– R –
`lifetime`	Manual memory management in all cases (Lifetime.1)	– R –
`lifetime`	Pointer/iterator dangling static analysis (Lifetime.1)	– R –
`lifetime`	Null checking (Lifetime.1)	F – C
`arithmetic`	Narrowing/lossy conversions implicit cases ([P3038R0] §12)	– R M –
`arithmetic`	Signedness promotions implicit cases ([P3038R0] §12)	– R M –
`arithmetic`	Lossy conversions via arithmetic overflow ([P3038R0] §12)	– – C

3 `type_safety` Profile

Enforce the [Pro.Type] safety Profile by default.

3.1 `reinterpret_cast` in all cases (Type.1.1)

For `reinterpret_cast`:

- R reject ("illegal cast, using `reinterpret_cast` requires `[[suppress(type_safety)]]`")

> **Note** This covers casting from a non-pointer to a pointer, which is also has bounds safety and lifetime safety implications. Writing `[[suppress(type_safety)]]` is required to opt out and enable a cast to a pointer, but that alone does not also opt out of bounds checks or null dereference checks. An subsequent additional `[[suppress(bounds_safety)]]` is required to opt out and enable arithmetic on the resulting pointer, and a subsequent additional `[[suppress(lifetime_safety)]]` is required to opt out of checking null dereference of the resulting pointer.

3.2 `const_cast` in all cases (Type.3)

For `const_cast`:

- if the source is const and the destination is not const (i.e., the cast definitely "casts away constness"), then R reject ("illegal cast, casting away `const` requires `[[suppress(type_safety)]]`")
- otherwise, if the source and the destination are both const (i.e., the cast definitely has no effect), then F M normatively encourage implementations to offer automatic source modernization to remove the redundant cast ("this cast has no effect and should be removed")
- otherwise (e.g., if the source is a deduced type and could be const or non-cosnt), F M normatively encourage implementations to offer automatic source modernization to change to `std::as_const` ("to explicitly add `const`, use `std::as_const`")

3.3 `static_cast` problematic cases (Type.1.2, Type.1.3, Type.1.4, Type.2)

For `static_cast<To>(from)` where `from` is of type `From`:

- if `From` and `To` are the same type, F normatively encourage implementations to offer automatic source modernization to replace the cast with `To{from}`
- otherwise, if implicit conversion from `From` to `To` is legal, F normatively encourage implementations to offer automatic source modernization to remove the cast
- otherwise, if the conversion from `From` to `To` is a built-in narrowing conversion, then R reject ("illegal narrowing conversion from `From` to `To` can lose information, using implicit narrowing requires `[[suppress(type_safety)]]` or `[[suppress(arithmetic_safety)]]`") , and R M normatively encourage implementations to offer automatic source modernization to use `narrow` instead ("use `narrow<Target>(source)` instead")

- otherwise, if `From` or `To` is an arithmetic type, then R reject ("illegal cast, casting between incompatible types requires `[[suppress(type_safety)]]`")
- otherwise, if `From` and `To` are pointer (or reference) types, designate the types they point (or refer) to as `Deref_From` and `Deref_To`
    - if `Deref_From` and `Deref_To` are unrelated types, then R reject ("illegal cast, casting unrelated types requires `[[suppress(type_safety)]]`")
    - otherwise, if `Deref_To` is derived from `Deref_From` (i.e., this is a static downcast)
        - if `Deref_From` is a polymorphic type, then F C perform a `dynamic_cast<To>(from)`, and F M normatively encourage implementations to offer automatic source modernization to `dynamic_cast` ("this cast should be spelled `dynamic_cast`")
        - otherwise, R reject ("illegal cast, a static pointer/reference downcast of non-polymorphic types requires `[[suppress(type_safety)]]`") and F M normatively encourage implementations to offer automatic source modernization to make the `~Deref_From` destructor virtual (if a user-defined one is present) or add a defaulted virtual `~Deref_From` destructor ("the base type destructor `~DerefFrom` should be virtual")

Notes A static pointer downcast could now result in a `nullptr`, which can be caught via `lifetime_safety` null dereference testing (in practice, null dereference is already commonly trapped; see note in §6.3).

A static reference downcast could now result in an exception.

Additionally, consider fixing or rejecting some cases where the `static_cast` has inexpensive-to-check undefined behavior.

The non-local modernization to make the base destructor virtual would work only for program types, not library types which the program can't change. So for now this modernization should only be offered for declarations that are in the program's own source tree if that can be reliably detected (either by the project environment, or by noting whether the declaration is in a header included with `" "` or is in the same module, for example).

3.4 `dynamic_cast` redundancy and performance (Type.1.3, Type.1.4)

For `dynamic_cast<To>(from)` where `from` is of type `From`, and both are pointer types (or both are reference types):

- if `From` and `To` are the same type or if implicit conversion from `From` to `To` is legal, F M normatively encourage implementations to offer automatic source modernization to remove the cast

As a normative C++ conformance requirement, if the `type_safety` Profile is enabled:

- F the C++ implementation must support `dynamic_cast` with its full semantics in all compilation modes, even when operated in a non-standard subset mode that may disable other RTTI features and information (e.g., `-fno-rtti` and `/GR-` shall not disable `dynamic_cast` if the `type_safety` Profile is enabled)

- F the C++ implementation must ensure that executing a `dynamic_cast` has a worst case of O(N) execution time, where N is the number of distinct base class subobjects in the most-derived object (e.g., an implementation shall not use an algorithm that contains nested loops)

Notes It is intentional to specify big-O in terms of execution time, not number of comparisons. The intent is to require a baseline quality of implementation (QoI) that guarantees no undesirable extra work is done, such as to perform a `strcmp`-style string name comparison to test each base class as the inheritance hierarchy is traversed.

It is understood that meeting these conformance requirements could require an ABI break.

If improving `dynamic_cast` in these ways is considered to have insurmountable barriers, as a fallback we could add a new cast.

3.5 `(c_style)cast` problematic cases (Type.4)

For a C-style cast `(To)from` where `from` is of type `From`:

- if the C-style cast would perform a `const_cast` (possibly following another cast), then apply the same rules as for `const_cast`
- if the C-style cast would perform a `static_cast` (possibly followed by a `const_cast`), then apply the same rules as for `static_cast`
- if the C-style cast would perform a `reinterpret_cast` (possibly followed by a `const_cast`), then apply the same rules as for `reinterpret_cast`

3.6 `functional_style(cast)` problematic cases (Type.4)

For a functional-style cast `To(from)` where `from` is of type `From`:

- if the functional-style cast would be equivalent to a `(c_style)cast`, then apply the same rules as for `(c_style)cast`
- otherwise, F perform a `functional_style{cast}`, and F M normatively encourage implementations to offer automatic source modernization to `functional_style{cast}` ("this cast should be spelled `C{args}`")

3.7 `union` in all cases (Type.7)

For a `union` object declaration:

- F C add a nonintrusive discriminator for the `union` object and inject a discriminator check at each use of a member of the union

Notes This is the most experimental/aggressive "F" and needs performance validation.

One possible F implementation is to have a global map of `void*` to `uintNN_t`, that externally stores every existing union object's address and current active member (where the number of alternatives fits into an NN-bit discriminator). Is this a potentially scary performance bottleneck? Maybe! But (a) the user asked for `type_safe` enforced mode, and (b) it's ABI-compatible and the user didn't have to change their code, we fixed it for them with just a recompile (and every violation is an actual bug,

and we have a good chance of finding at least one bug, which makes users much more willing to accept the change). To opt out, such as for a `union` variable known to be used for intentional bit-swizzling or used safely in a tagged struct hierarchy, use `[[suppress(type_safety)]]` on the `union` variable's declaration. — I do expect a lively discussion, feedback welcome!

Why not suggest an M to offer to rewrite `union` to `variant`? I have two concerns: (1) `variant` is safe but is not sufficiently functional to be a complete replacement for `union` (e.g., it does not create a unique type). (2) Offering the rewrite requires access to all source code uses of the declared union object across the project, which is difficult even within a translation unit and impossible for a union object in a header shared beyond the current project. (A third but not compelling reason is that a `variant` is more awkward to use, but that is not a showstopper.)

3.8 `va_arg` in all cases (Type.8)

For `va_arg` in a function definition:

- R reject ("type-unsafe operation, using `va_arg` requires `[[suppress(type_safety)]]`"), and R M normatively encourage implementations to offer automatic source modernization to variadic parameters instead ("use variadic parameters instead")

Note If the function definition is not in a header file, the M implementation would also need to be able to change forward declarations of this function in header files. So for now this modernization should only be offered for declarations that are in the program's own source tree if that can be reliably detected (either by the project environment, or by noting whether the declaration is in a header included with `" "` or is in the same module, for example).

4 **bounds_safety** Profile

Enforce the [Pro.Bounds] safety Profile by default, and guarantee bounds checking when `size`/`ssize` is available.

4.1 Pointer arithmetic in all cases (Bounds.1, Bounds.3)

For every use of an object `p` of raw pointer type:

- if the use applies any operator to `p` other than `*p`, `p->`, or `&p`, then R reject ("bounds-unsafe operation, performing pointer arithmetic or bitwise manipulation requires `[[suppress(bounds_safety)]]`")

Notes I think it is clearer to state a positive "allow-list" of operators intended to be allowed. The following alternative negative "deny-list" formulation is probably equivalent, but I think less clear, and it has the "did we forget any" problem:

> if the use is `p[`*`expr`*`]`, *`expr`*`[p]`, `++p`, `p++`, `p +` *`expr`*, `p +=` *`expr`*, `--p`, `p--`, `p -` *`expr`*, `p -=` *`expr`*, `~p`, `p ^` *`expr`*, `p ^=` *`expr`*, `p &` *`expr`*, `p &=` *`expr`*, `p |` *`expr`*, or `p |=` *`expr`*
>
> Low-level code that heavily relies on pointer arithmetic will want to `[[suppress(bounds)]]` on a larger region, such as a class or source file. Alternatively, we could explore adding a larger pointer type that includes tags and supports safe arithmetic, as a drop-in replacement (but with ABI impact).

4.2 Array-to-pointer decay in all cases (Bounds.3)

For every use of an array name `a`:

- if the use would decay to a pointer, then R reject ("bounds-unsafe operation, array-to-pointer decay requires `[[suppress(bounds_safety)]]`")

Note The primary reason for Bounds.3 that rule is that pointers should point to single objects and pointer arithmetic should be avoided, which are already enforced by the previous rule. This rule is only needed because of interop with non-bounds-safe code, so that an array is name cannot be silently passed to bounds-unsafe code that could attempt to perform arithmetic.

4.3 Subscript checking including arrays/`vector`/`span`/etc. (Bounds.4)

For every valid expression of the form `a[b]`:

- if `b` evaluates to an integral index, and `a` is a sequence where `std::size(a)` and `std::ssize(a)` are available, and `std::begin(x)+2` is a valid expression (i.e., this is a contiguous sequence):
  - then C perform a bounds check as-if a `contract_assert<Bounds>(0 <= b && b < max_size)`, where `max_size` is either `std::ssize(x)` or `std::size(x)` depending if `b` is signed or unsigned respectively, and each of `a` and `b` is evaluated only once

Note Integrated with C++26 contracts; this should follow whatever is approved for that, but we do need the contract group/category extension so that we can have customized global handler specifically for Bounds checks.

If a violation happens, some programs will want to terminate (the recommended default), others will want to log-and-continue, throw an exception, integrate with a project-specific critical fault infrastructure, etc. The action taken can be customized using a global `Bounds` violation handler, such as like this (strawman syntax):

```
std::Bounds.set_handler(
    [](std::string_view violation_message) {

        //  Arbitrarily customized handling can be installed here
        CallMyInHouseFramework( violation_message, other_info );
        throw My::Bespoke::RollbackException( additional, data );

    }
);
```

Note Some projects strictly require a given handling. For example, an OS component may require termination and that no user code be allowed to run after a bounds violation; whereas a business application may require log-and-continue using its custom error reporting telemetry. Each program can enforce such a rule for itself by enforcing which global handler is used (or none, so that the default is used).

Injecting bounds checks at call sites deliberately avoids implementing bounds-checking intrusively for each individual container/range/view type. Implementing bounds-checking non-intrusively and automatically at the call site makes full bounds checking available for every existing standard and user-written container/range/view type out of the box: Every subscript into a `vector`, `span`, `deque`, or similar existing type in third-party and company-internal libraries would be usable in "`bounds_safe`" mode without any need for a library upgrade.

It's important to add automatic call-site checking now before libraries continue adding more subscript bounds checking in each library, so that we avoid duplicating checks at the call site and in the callee. As a counterexample, C# took many years to get rid of duplicate caller-and-callee checking, but succeeded and .NET Core addresses this better now; we can avoid most of that duplicate-check-elimination optimization work by offering automatic call-site checking sooner.

Note Efficient bounds checking requires known optimizations such as that redundant checks be hoisted out of loops. In cases that are by-construction, such as range-`for`, the front-end could have enough information to hoist the redundant checking. Otherwise, the middle-end or back-end would hoist the redundant checking.

5 `initialization_safety` Profile

Enforce the [Pro.Type] Type.5 and Type.6 initialization-before-use rules by default.

5.1 Member variables (Type.6)

In a constructor for a class with a nonstatic data member variable `M`:

- if `M` is not is not initialized by default-initialization and has no initializer, then R reject ("uninitialized member variable, leaving a data member uninitialized requires `[[suppress(initialization_safety)]]`")

5.2 Non-member variables (Type.5)

In a declaration for a local variable `V`:

- if `V` is not is not initialized by default-initialization and has no initializer, then R reject ("uninitialized local variable, leaving a variable uninitialized requires `[[suppress(initialization_safety)]]`")

Notes I start with this widely accepted rule to improve the chances of getting consensus in WG21.

In general, for acceptable performance an implementation must be able to avoid initialization of local variables that are not accessed on an early-exit path from the function, when the initialization can be known not to have side effects.

This is orthogonal to proposals to "F"ix by forcing zero-initialization, such as P2723 and P2795. However, there are serious concerns about forcing zero-initialization: Security experts have long pointed out that while doing so does "fix" some bugs, it actively masks other bugs (e.g., where zero is not a valid safe value for that object so there is still a bug, and forcing the initialization to occur anyway actively hides the problem from sanitizers because the object appears to have been initialized). Anything we do in that direction should address that concern. Also, attempts to adopt a forced zero-initialization rule in large code bases (e.g., Windows) have failed due to performance issues.

This leaves the door open for a later separate followup paper to propose initialization-before-use (example diagnostic; definite-first-use rules) as a possible compatible relaxation. That belongs in a separate followup paper, but I mention it for completeness because that approach has advantages:

> it is more flexible (e.g., it enables different constructors to be used with immediately-invoked lambda trick), and by making it easier to explicitly provide program-meaningful initial values in more cases, it reduces the desire to force zero-initialization which has the drawbacks mentioned earlier in this note;
>
> it can be more efficient, by avoiding "dummy" construction when a program-meaningful value is not yet available at the point the variable is declared;
>
> it has successful precedent in other widely-used languages where it is known to be easy to use (e.g., C#, Ada); and
>
> it is simple to specify (briefly: between the variable declaration and a definite first use, there must be an initialization, there must not be a loop, and if there is a branch that branch must either initialize on both branch paths (in which is counts as the initialization) or neither, and the same rule replied recursively to nested-branches).

6 `lifetime_safety` Profile

Enforce the [Pro.Lifetime] safety Profile by default, ban manual dynamic lifetime management by default, and guarantee null checking.

6.1 Manual memory management in all cases (Lifetime.1)

If an expression contains `new`, `delete`, `delete[]`, `malloc`, or `free`, including placement `new`:

- then R reject ("illegal manual memory allocation/deallocation, using `new`, `delete`, `malloc`, or `free`, requires `[[suppress(lifetime_safety)]]`")

6.2 Pointer/iterator dangling static analysis (Lifetime.1)

Implement the scalable linear analysis rules of the Guidelines' [Pro.Lifetime] static analysis, which covers common cases of pointer/iterator/view/range dangling (e.g., covers most cases of using an invalidating iterator inside a loop that modifies the collection the iterator refers to).

For every valid expression of the form `*p` or `p->` or `p[]`, for a variable `p` of generalized Pointer type (includes iterators, etc.):

- if at the point of that expression, `p` is *invalid*, then R reject ("`p` is invalid, dereferencing an invalid `p` requires `[[suppress(lifetime_safety)]]`")
- otherwise, if at the point of that expression, `p` is *definitely null*, then R reject ("`p` is null, dereferencing a null `p` requires `[[suppress(lifetime_safety)]]`")
- otherwise, if at the point of that expression, `p` is *possibly null*, then R reject ("`p` could be null, dereferencing a possibly null `p` requires `[[suppress(lifetime_safety)]]`")

Note As described in the analysis specification, implementations are encouraged, but not required, to show the exact source code line that invalidated or nulled `p`. Tracking this additional information does not add time complexity to the analysis.

6.3 Null checking (Lifetime.1)

For every valid expression of the form `*p` or `p->`:

- if `p == nullptr` is valid and returns a boolean-testable result:
  - then C perform a null check as-if a `contract_assert<Lifetime>(p != TYPEOF(p){})`, and `p` is evaluated only once

Notes Some C++ features, such as `delete`, have always done call-site null checking.

Integrated with C++26 contracts; this should follow whatever is approved for that, but we do need the contract group/category extension so that we can have customized global handler specifically for Bounds checks.

The compiler could choose to not emit this check (and not perform optimizations that benefit from the check) when targeting platforms that already trap null dereferences, such as platforms that mark low memory pages as unaddressable.

If a violation happens, some programs will want to terminate (the recommended default), others will want to log-and-continue, throw an exception, integrate with a project-specific critical fault infrastructure, etc. The action taken can be customized using a global `Lifetime` violation handler, such as like this (using the contracts feature extended with contract groups/categories) (strawman syntax):

```
std::Lifetime.set_handler(
    [](std::string_view violation_message) {

        //  Arbitrarily customized handling can be installed here
        CallMyInHouseFramework( violation_message, other_info );
        throw My::Bespoke::RollbackException( additional, data );

    }
);
```

Note Some projects strictly require a given handling. For example, an OS component may require termination and that no user code be allowed to run after a null dereference attempt; whereas a business application may require log-and-continue using its custom error reporting telemetry. Each program can enforce such a rule for itself by enforcing which global handler is used (or none, so that the default is used).

Injecting null checks at call sites deliberately avoids implementing null-checking intrusively for each individual type. Implementing null-checking non-intrusively and automatically at the call site makes full null checking available for every existing standard and user-written pointer type out of the box: Every dereference of a `unique_ptr`, `shared_ptr`, `observer_ptr`, or similar existing type in third-party and company-internal libraries would be usable in "`lifetime_safe`" mode without any need for a library upgrade.

7 `arithmetic_safety` Profile

Enforce no data loss by default, by banning lossy conversions.

7.1 Narrowing/lossy conversions implicit cases ([P3038R0] §12)

If an expression contains an implicit conversion from an object `from` object of type `From`, to a target type `To`:

- if the conversion from `From` to `To` is a built-in narrowing conversion, then R reject ("illegal narrowing conversion from `From` to `To` can lose information, using implicit narrowing requires `[[suppress(type_safety)]]` or `[[suppress(arithmetic_safety)]]`") , and R M normatively encourage implementations to offer automatic source modernization to use `narrow` instead ("use `narrow<Target>(source)` instead")
- otherwise, if `To` is an accessible base class of `From` and the conversion is a derived-to-base copy or move operation (i.e., "slicing"), and `From` has more non-static data members than `To`, then R reject ("illegal derived-to-base slicing conversion loses information, using derived-to-base slicing requires `[[suppress(arithmetic_safety)]]`")

Note The second bullet is expressed in terms of the number of non-static data members, not `sizeof` which could be the same even if there are additional data members.

7.2 Signedness promotions implicit cases ([P3038R0] §12)

If an expression contains an implicit conversion for a `source` value of built-in type `Source`, to a built-in `Target` type with different signedness:

- then R reject ("implicit conversion from signed to unsigned (or unsigned to signed) can lose information, using implicit narrowing requires `[[suppress(arithmetic_safety)]]`") , and R M normatively encourage implementations to offer automatic source modernization to use `narrow` instead ("use `narrow<Target>(source)` instead")

7.3 Lossy conversions via arithmetic overflow ([P3038R0] §12)

If an expression contains an arithmetic operation that could encounter modular arithmetic overflow or underflow:

- then C perform an overflow/underflow check, and treat overflow/underflow as a `contract_assert<Value>(false)`

Note Integrated with C++26 contracts; this should follow whatever is approved for that, but we do need the contract group/category extension so that we can have customized global handler specifically for Bounds checks.

If a violation happens, some programs will want to terminate (the recommended default), others will want to log-and-continue, throw an exception, integrate with a project-specific critical fault infrastructure, etc. The action taken can be customized using a global `Value` violation handler, such as like this (strawman syntax):

```
std::Arithmetic.set_handler(
    [](std::string_view violation_message) {
```

```
            //  Arbitrarily customized handling can be installed here
            CallMyInHouseFramework( violation_message, other_info );
            throw My::Bespoke::RollbackException( additional, data );

        }
    );
```

Note Some projects strictly require a given handling. Each program can enforce which global handler is used for arithmetic violations (or none, so that the default is used).

7.4 Addition: `unchecked_narrow_cast` and `narrow`

Add [GSL] explicit `narrow` and `narrow_cast`, to enforce explicit narrowing conversions.

I recommend naming these `std::unchecked_narrow_cast` and `std::narrow`.

For a source value `from` of type `From`, and a target type `To`, where either:

- both are built-in non-`char` arithmetic types (i.e., explicit size reduction and/or sign change), or
- `To` is an accessible base class of `From` (i.e., explicit slicing),

then:

- `unchecked_narrow_cast<To>(from)` performs today's unsafe `static_cast<To>(from)`. The latter still cannot be written under the `type_safety` Profile.
- `narrow<To>(from)` safely performs today's `static_cast<To>(from)` by checking if the result `== from` without signedness promotions. If yes, it returns the result. Otherwise, it throws a `std::narrowing_error`.

8 Adoptability and impact: Broadly applicable strict mode

Not all applications will be able to use all rules. That's why we have granular opt-ins and opt-outs.

However, *most* code will want to use the broadly applicable safety rules by default. And this is the existing prior art in competing languages: to have a common set of safety rules on by default.

To maximize adoptability, it must be as easy as possible to enable all the common and broadly useful safety rules by default. That's why we need a single broadly useful common/default safety Profile ('one big red button', or OBRB) that enforces common widespread best practices in one step. Just as users can compile C++ code and specify a standard mode simply (e.g., `-std=c++23` or `/std:c++26`), they could as easily be able to specify a "strict mode" of each standard (e.g., request "`C++26.strict`" that is initially a synonym for the union of all Profiles in this paper). This subset could evolve, to add additional guarantees as desired in `C++29.strict` mode and so on. Enforcing a Profile still allows using all of C++, but requires explicit "break-the-glass, trust me" opt-out for code the language cannot efficiently guarantee is safe

9 Addition: Null-terminated `zstring_view`

This is one of the commonly-requested features from the [GSL] library that does not yet have a `std::` equivalent. It was specifically requested by several reviewers of this work.

This section suggests adding a corresponding type for each `std::@string_view`:

- `basic_zstring_view`
- `zstring_view`
- `zwstring`
- `zu8string`
- `zu16string`
- `zu32string`

The specification for each `std::@zstring_view` is identical to that of the corresponding `std::@string_view` except:

(1) It is null-terminated, and the final null does not count in the size. Corollary: It does not have suffix-shortening operations.

- `size()` and `length()` are guaranteed to equal `strlen(data())`

> **Note** I don't know of a reason to allow `operator[]` to access the null terminator. If there are use cases where that is needed, then `operator[](size())` can be made valid (i.e., be specified to not be diagnosed as a bounds violation) and be guaranteed to be `'\0'`.

- `remove_suffix()` is removed
- `substr()`'s second parameter `count` is removed and the function behaves as-if `count==npos`, i.e., it returns a substring for the rest of the string

> **Note** For usability, it might be nice to add a `last(size_type count)` or `suffix(size_type count)` for both `@zstring_view` and `@string_view`. And, correspondingly, a `first(size_type count)` or `prefix(size_type count)` for `@string_view`.

(2) It does not have the bounds-unsafe `copy` member function.

- `copy(CharT*, size_type, size_type)` is not available

(3) It has an implicit `operator std::@string_view()` conversion operator that returns a `@string_view` that does not include the null terminator, and a `to_@string_view_with_null()` named conversion function that returns a `@string_view` that does include the null terminator.

(4 – optional) It is not read-only, including that `std::copy` and similar algorithms can be used to modify its contents. Therefore add `c`-prefixed versions of each of the above to signify `basic_zstring_view<const CharT>`; for example, `czstring_view` is an alias for `basic_zstring_view<const char>`.

10 References

[P2816R0] B. Stroustrup and Gabriel Dos Reis. “Safety Profiles: Type-and-resource safe programming in ISO standard C++” (WG21 paper and SG23/EWG presentation, February 2023).

[P2687R0] B. Stroustrup and Gabriel Dos Reis. “Design alternatives for type-and-resource safe C++” (WG21 paper, October 2023).

[P3038R0] B. Stroustrup. “Concrete suggestions for initial Profiles” (WG21 paper, December 2023).

[Pro.Type] B. Stroustrup and H. Sutter, editors. *C++ Core Guidelines* Pro.Type Profile for type safety and initialization safety.

[Pro.Bounds] B. Stroustrup and H. Sutter, editors. *C++ Core Guidelines* Pro.Bounds Profile for bounds safety.

[Pro.Lifetime] B. Stroustrup and H. Sutter, editors. *C++ Core Guidelines* Pro.Lifetime Profile for lifetime safety.

[GSL] B. Stroustrup and H. Sutter, editors. *C++ Core Guidelines* Guideline Support Library. It is by design that the Pro.* safety Profiles and GSL are immediately adjacent to each other.

[Chisnall2024] D. Chisnall. Reply to “Re: The Case for Rust (in the base system)” (freebsd-hackers list, January 2024).