Core safety profiles for C++26

Document Number: **P3081 R2**
Date: 2025-02-04
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: EWG, LEWG

Contents

1 Overview: Motivation in a nutshell 2
2 Approach: Strategy and tactics 4
3 General profiles support 7
4 **`type`** profile 12
5 **`bounds`** profile 18
6 **`lifetime`** profile 21
7 **`arithmetic`** profile.... 23
8 Related papers / extension feedback.... 24
9 References.... 25

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 enforced safety profile rules, in addition to any other profile rules and 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 or mechanically updating the code where possible (i.e., no manual code changes required, such as by adding subscript checking) 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. (As of R2 this has been deferred to a future separate paper.)

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 bounds, type (including initialization), and lifetime safety. Initially targeting those four areas can significantly reduce security vulnerabilities (CVEs), after which we should also address further areas.

As elaborated in "C++ safety, in context," 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++.

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.

1.4 Revision history

R2: Added wording in [except.spec] that exceptions thrown from contract assertion evaluation do not affect the `noexcept` operator. Added `noexcept` to `index_in_range`. Adopted some wording suggestions proposed in [P3589R0]. Implemented EWG telecon feedback to have each profile rules section retain motivation and design rather than wording only. Added section 9 to refer to other extensions proposed in [P3589R1] and [P3471R2]. Implemented EWG telecon feedback to separate out modernizations to a follow-on paper. Restored inadvertent omission from R0 of local variable initialization. Remove optional `zstring_view` part of the proposal as it's optional and we did not get to it in telecon reviews between meetings, it can be proposed separately. Implemented SG21 telecon clarification suggestions.

R1: Applied SG23 and EWG direction from Wrocław. Merged initialization safety profile into type safety. Deferred controversial parts (e.g., no longer includes implementing `static_cast` as `dynamic_cast`, or requiring `dynamic_cast` performance guarantees) based on Wrocław and post-Wrocław EWG telecon feedback. Applied various improvements (e.g., allowing `reinterpret_cast` to `std::byte` and from pointer to `uintptr_t`). Split off the non-local static lifetime analysis into [P3465R1] as SG23 encouraged. Added proposed wording.

1.5 Acknowledgments

Special thanks to Bjarne Stroustrup for driving the initiative for profiles in standard C++, and to Jens Maurer for his essential assistance with many rounds of refinement to the proposed standards wording.

Many thanks also to all the experts whose feedback on drafts of this material has been invaluable, including: Gašper Ažman, Jean-François Bastien, Sean Baxter, Joshua Berne, Joe Bialek, Frank Birbacher, Jonathan Caves, Gabriel Dos Reis, Timur Doumler, Daniel Frampton, Tanveer Gani, Daniel Griffing, Russell Hadley, Mark Hall, Tom Honermann, Michael Howard, Oliver Hunt, Corentin Jabot, Loïc Joly, Lisa Lippincott, Ulzii Luvsanbat, Rico Mariani, Greg Marr, Chris McKinsey, Bogdan Mihalcea, Jonathan Müller, Roger Orr, Robert Seacord, Jan Schultke, Mads Torgersen, Guido van Rossum, Ville Voutilainen, Roy Williams, Michael Wong. Thanks also to all the other WG21 participants and other experts I have forgotten to personally name; your feedback is appreciated.

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 `bounds`, `type`, and `lifetime`. 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, [Chisnall 2024]:

> *"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 with a profile `P` enforced, use `[[profiles::suppress(std::P)]]` to say "trust me" and opt out of profile `P` in that scope.

Opt-out is granular. We allow suppressing a profile on a statement, including a compound-statement (block scope), and allow enabling a profiles on a translation unit and via implementation-defined means (e.g., command line switch). For example, when this code is compiled with the `type` profile enforced, it has the commented compile-time meaning:

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

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

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` and `bounds`" mode, it has the commented compile-time meaning, because the cast is disallowed in the `type` profile and the pointer arithmetic is disallowed in the `bounds` profile:

```
{
    ++(double*)&i;  // error: type+bounds

    [[profiles::suppress(std::type)]]
    ++(double*)&i;  // error: bounds

    [[profiles::suppress(std::type)]]
    [[profiles::suppress(std::bounds)]]
    ++(double*)&i;  // ok

}
```

```
{
    [[profiles::suppress(std::type)]] {

        ++(double*)&i;  // error: bounds

        [[profiles::suppress(std::bounds)]]
        ++(double*)&i;  // ok
    }
}
```

2.2 Tactics

We can address each rule violation using any of three basic tactics:

	Tactic	Adoptability / UX	Manual code changes?
Fix	**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	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	Where necessary, diagnose violations at run time, and let the program customize how violations are handled (ideally the same as contract group violation handlers, once those are available)	Automatically diagnoses bugs in existing code just by recompiling the code	None ✓

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 `std::` level for a given profile `P`	**What parts of `P` are enabled**	**Adoptability**	**Safety improvement**
`suppress(P)`	None	n/a — ordinary no-guardrails C++	None
`apply(P)`	Fix ✓ Check ✓ Modernize ✓ QoI: Optionally also 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

Following EWG 2025-01-22 telecon direction, modernizations will be covered in a follow-on paper.

3 General profiles support

Change [intro.compliance.general] paragraph 1 as follows:

The set of *diagnosable rules* consists of all syntactic, ~~and~~ semantic, and profile rules in this document except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in “undefined behavior”.

Change [intro.compliance.general] paragraph 2.4-2.5 as follows:

Furthermore, a conforming implementation shall not accept

- a preprocessing translation unit containing a #error preprocessing directive ([cpp.error]), ~~or~~
- a translation unit with a static_assert-declaration that fails ([dcl.pre]), or
- a translation unit with a profile-rejected diagnostic ([dcl.attr.profile]).

At the end of [dcl.attr.grammar] paragraph 1, add new productions:

profile-name:
    attribute-token

The identifiers in a *profile-name* or *profile-argument* are not looked up ([basic.lookup]).

At the end of [dcl.attr], add a new section after [dcl.attr.nouniqueaddr]:

x.x.13 Profile attributes [dcl.attr.profile]

A *profile attribute* is an attribute where the *attribute-token* is `profiles::enforce`, `profiles::apply`, or `profiles::suppress` followed by an *attribute-argument-clause* of the form (*profile-name*). The attribute-token `profiles::suppress` may appear on a statement. A profile attribute whose *attribute-token* is `profiles::enforce` or `profiles::apply` may be applied only to *empty-declaration*, and that empty-declaration shall be the lexically first *declaration* (if any) in the translation unit.

For a translation unit where neither `profiles::enforce(P)` nor `profiles::apply(P)` was written on an empty-declaration, profiles defaults that behave as-if `profiles::enforce(P)` or `profiles::apply(P)` had been written on an empty-declaration may be defined in an implementation-defined manner. *[Note:* It is unspecified whether an implementation provides a way to distinguish whether such a default profile attribute applies to: the entire translation unit; to code lexically in the C++ source file; and/or also to code in some but not all included header files (for example, to code in header files that are considered part of a local project but not to code that are considered part of third-party library header files). *– end note]*

The *profile-name* in a profile-designator identifies a *profile*, which is a set of language restrictions applied after translation phase 7. *[Note:* Enforcing or applying a profile can add additional behavior to the program’s code; can constrain undefined, unspecified, and erroneous behavior; or can do both. For example, enforcing or applying the `std::bounds` profile adds a subscript bounds check and changes a subscript out-of-bounds access from undefined behavior into a defined behavior. However, enforcing or applying a profile does not change the semantics of the program’s code, such as the results of overload resolution or whether special member functions are deleted. *– end note] [Example:*

```
template<class T>
int f(T x , T * = const_cast<T*>( (const T*)nullptr )) { return 1; }

int f(...) { return 0; }

int x = f(42);
```

If the profile `std::type` is not enforced, x equals 1. If the profile `std::type` is enforced, the implementation does not accept the translation unit. A program in which x equals `0` is never produced. *– end example]*

The profiles listed in table X are specified in this document. Enforcing or applying profile `std::strict` is equivalent to enforcing or applying all of profiles `std::type`, `std::bounds`, and `std::lifetime` individually.

Table X: Profiles summary [tab:profiles.summary]

profile-name	**Subclauses**
`std::type`	[expr.reinterpret.cast] , [expr.const.cast], [expr.static.cast], [expr.dynamic.cast], [class.base.init], [cstdarg.syn], [class.union.general]
`std::bounds`	[expr.add], [expr.sub], [expr.pre.incr], [expr.post.incr], [expr.pre.ass], [conv.array]
`std::lifetime`	[expr.delete], [c.malloc], [expr.unary.op]
`std::arithmetic`	[dcl.init.list], [conv.integral], [expr.pre]

At a program point Q, a *profile* P is *suppressed* if any enclosing scope of Q has the profile attribute `profiles::suppress(P)`. Recommended practice: Implementations should emit a diagnostic if such a scope contains no construct that would be profile-rejected by profile P.

At a program point Q, a *profile* P is *enforced* if it is not suppressed and: the profile attribute `profiles::enforce(P)` has been encountered, or the default profile attribute (if any) for the translation unit is equivalent to `enforce(P)`.

At a program point Q, a *profile* P is *applied* if it is not suppressed and: the profile attribute `profiles::apply(P)` has been encountered, or the default profile attribute (if any) for the translation unit is equivalent to `apply(P)`.

At a program point Q, a *profile-name* P is *enabled* if it is either enforced or applied.

[Example: A

```
// file sample.cpp

[[profiles::apply(std::type)]];    // for this translation unit, profile std::type
                                   // is applied everywhere below except where
                                   // noted otherwise

void f( B* pb ) {
```

```
        [[profiles::suppress(std::type)]]   // for this statement,
        static_cast<C*>(pb);                // std::type is suppressed

        [[profiles::suppress(std::type)]] { // for this compound-statement,
            static_cast<C*>(pb);            // std::type is suppressed
        }

    }
```

– end example]

A program construct C at program point Q being *profile-rejected by profile P* means:

- If C is
  - potentially evaluated ([basic.def.odr]) ,
  - outside a requires expression ([expr.prim.req]),
  - outside a discarded statement ([stmt.if]), and
  - outside of a candidate function template specialization ([over.match.funcs.general]) that is not named by an expression ([basic.def.odr])

  then:
  - If C appears in the body of a function that is defined in this document, or if it was generated by the expansion of a macro that is defined in this document, the rejection is ignored. *[Note:* For example, if an implementation implements the `offsetof` defined in this document as a macro whose replacement results in inserting a `reinterpret_cast`, and that cast would be profile-rejected in that position had it appeared textually in the source file, the `reinterpret_cast` is considered part of the implementation and will not result in a diagnostic. *– end note]*
  - Otherwise, if P is enforced at Q, the implementation shall emit a *profile-rejected diagnostic* that C is not allowed when profile P is enforced.
  - Otherwise, if P is applied at Q, P is erroneous behavior and the implementation may emit a diagnostic that C is discouraged. Recommended practice: Implementations should emit a diagnostic only if that will not stop translation, such as if warnings are not being treated as errors.
- *[Note:* Otherwise, the rejection is ignored. *– end note]*

Change [dcl.pre] paragraph 1's *empty-declaration* production as follows:

empty-declaration:
  attribute_specifier_seq$_{opt}$;

Change [dcl.pre] paragraph 15 as follows:

An *empty-declaration* has no effect. The optional *attribute-specifier-seq* appertains to the *empty-declaration.*

Change [except.spec] paragraph 5 bullet 2 as follows:

- *E* implicitly invokes a function (such as an overloaded operator, an allocation function in a *new-expression*, a constructor for a function argument, or a destructor) that is not a contract violation handler or implicit contract assertion, and that has a potentially-throwing exception specification, or

and change bullet 6 as follows:

- any of the immediate subexpressions of *E* that is not a contract violation handler or implicit contract assertion is potentially-throwing.

3.1 If P2900 is merged first

The design is a profile check emits a `contract_assert` as if written by the programmer. The programmer has the same configuration options as a `contract_assert` written by the user.

same configuration options as contract_assert written by user

At the end of [dcl.attr.profile], also add:

A condition C at program point Q being *profile-checked by profile P* means:

- If P is enabled at Q, the implementation shall add an *implicit contract assertion* to evaluate C with `assertion_kind::implicit` and `detection_mode` value corresponding to P. The evaluation semantic of a contract assertion introduced by a profile is implementation defined. Recommended practice: If P is enforced at Q, implementations should use the evaluation semantic `quick_enforce` or `enforce`, and emit a diagnostic if the evaluation semantic used in execution may be `observe` or `ignore`. If P is applied at Q, implementations should use the evaluation semantic `observe`, and emit a diagnostic if the evaluation semantic used in execution may be `ignore`.

In [support.contracts.kind], add the bullet:

- `assertion_kind::implicit`: the evaluated contract assertion was an implicit contract assertion [(dcl.attr.profile]).

Change [support.contracts.detection] as follows:

Enum class detection_mode [support.contracts.detection]

The type `detection_mode` specifies the manner in which a contract violation was identified ([basic.contract.eval]). Its enumerated values and their meanings are as follows:

— `detection_mode::predicate_false`: the contract violation occurred because the predicate evaluated to `false` or would have evaluated to `false`.

— `detection_mode::evaluation_exception`: the contract violation occurred because the evaluation of the predicate evaluation exited via an exception.

— `detection_mode::type`: the contract assertion was evaluated as part of the `std::type` profile.

— `detection_mode::bounds`: the contract assertion was evaluated as part of the `std::bounds` profile.

— `detection_mode::lifetime`: the contract assertion was evaluated as part of the `std::lifetime` profile.

Recommended practice: Implementation-defined enumerators should have a name that is an identifier reserved for the implementation ([lex.name]) and a minimum value of `1000`.

3.2 If P2900 is not merged first, use this until P2900 is available

At the end of [dcl.attr.profile], also add:

A condition C at program point Q being *profile-checked by profile P* means:

- If P is enabled at Q, C is evaluated and this evaluation is an *implicit contract assertion*; if it does not evaluate to true, then `std::profile_violation(PP)` is invoked where PP is the `detection_mode` value corresponding to P.

If no replacement function for `std::profile_violation` is provided by the program ([replacement.functions]), the default behavior of `std::profile_violation` with any argument is to terminate in an implementation-defined manner. If a profile-check would cause termination, it is implementation-defined whether `std::profile_violation` is called.

In [replacement.functions], add a new paragraph:

A C++ program may provide the definition of the following function signature declared in header `<debugging>`:

```
void std::profile_violation( std::detection_mode );
```

In [replacement.functions], modify paragraph 4 as follows:

The program's definitions are used instead of the default versions supplied by the implementation ([new.delete]) ([dcl.attr.profile]). Such replacement occurs prior to program startup ([basic.def.odr], [basic.start]). The program's declarations shall not be specified as inline. No diagnostic is required.

After [???], add this new section:

Enum class `detection_mode` [support.contracts.detection]

The type `detection_mode` specifies the manner in which a contract violation was identified ([basic.contract.eval]). Its enumerated values and their meanings are as follows:

— `detection_mode::type`: the contract assertion was evaluated as part of the `std::type` profile.

— `detection_mode::bounds`: the contract assertion was evaluated as part of the `std::bounds` profile.

— `detection_mode::lifetime`: the contract assertion was evaluated as part of the `std::lifetime` profile.

4 **type** profile

Enforce the [Pro.Type] safety rules.

4.1 `reinterpret_cast` (Type.1.1)

From [Pro.Type]

Type.1: Avoid casts

1.1: Don't use reinterpret_cast

For `reinterpret_cast` that is not a cast to `std::byte`, and not a cast from a pointer to `uintptr_t`:

- R reject

Expand [expr.reinterpret.cast] paragraph 1 as follows:

> The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`.
>
> If T is not a pointer or reference to *cv* `std::byte`, and either v is not a pointer or T is not `uintptr_t`, the expression is profile-rejected by profile `std::type` ([dcl.attr.profile]).
>
> If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue, array-to-pointer, and function-to-pointer standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

Note This covers casting from a non-pointer to a pointer, which also has bounds safety and lifetime safety implications. Writing `[[profiles::suppress(std::type)]]` 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. A subsequent additional `[[profiles::suppress(std::bounds)]]` is required to opt out and enable arithmetic on the resulting pointer, and a subsequent additional `[[profiles::suppress(std::lifetime)]]` is required to opt out of checking null dereference of the resulting pointer.

4.2 `const_cast` (Type.3)

From [Pro.Type]

Type.3: Don't use const_cast to cast away const (i.e., at all)

For `const_cast`:

- If the cast casts away constness, then R reject

Note [Pro.Type], which this core set of rules is based on does not mention volatile, and so this section also does not mention volatile.

Expand [expr.const.cast] paragraph 1 as follows:

> The result of the expression `const_cast<T>(v)` is of type `T`.
>
> If casting away constness, the expression is profile-rejected by profile `std::type`.
>
> If `T` is an lvalue reference to object type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue, array-to-pointer, and function-to-pointer standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `const_cast` are listed below. No other conversion shall be performed explicitly using `const_cast`.

4.3 `static_cast` (Type.1.2, Type.1.3, Type.1.4, Type.2)

From [Pro.Type]

> Type.1: Avoid casts
>
> > 1.2: Don't use static_cast for arithmetic types
> >
> > 1.3: Don't cast between pointer types where the source type and the target type are the same
> >
> > 1.4: Don't cast between pointer types when the conversion could be implicit
>
> Type.2: Don't use static_cast to downcast: Use dynamic_cast instead

For `static_cast<To>(from)`:

- If the conversion from `from` to `To` is a built-in narrowing conversion to a type other than `bool`, then R reject, and encourage implementations to suggest using `narrow` instead
- otherwise, if this is a downcast (pointer or reference cast from base to derived), then R reject.

Provide a `narrow<To>(from)` that follows [GSL]. `narrow` does not need to be SFINAE-friendly.

In [utility], add a new subclause as follows:

> **x.x.10 narrow [utility.narrow]**
>
> ```
> template<class To, class From>
>   constexpr To narrow(From const& from);
> ```
>
> > *Mandates:* `To` is not a reference type, and the argument to `from` is not a constant-expression, and `is_constructible_v<To, From>` is true, and either converting `from` to `To` is an implicit conversion ([over.best.ics.general]) or `To` is a base class of `From`.

Effects: If `static_cast<T>(from) == from` is valid, does not perform promotions that change sign ([conv.prom]), and evaluates to true, returns `static_cast<To>(from)`. Otherwise, throws an exception of type `std::bad_cast`.

Note This goes beyond `std::in_range` (C++20) and `std::saturate_cast` (C++26) which are designed for integer types, and do not cover other built-in narrowing (e.g., `float` to `int`) or slicing (copyable derived to base).

Expand [expr.static.cast] paragraph 1 as follows:

The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`.

If T is not void, then let `From` be the type of v and:

- If the conversion from v to T is a narrowing conversion ([dcl.init.list]) and T is not `bool`, the expression is profile-rejected by profile `std::type` ([dcl.attr.profile]). Recommended practice: Implementations should emit a diagnostic that suggests using `std::narrow<T>(v)` instead ([utility.narrow]).
- Otherwise, if `From` and T are both pointer types then let `Deref_From` and `Deref_T` be the types they respectively point or refer to, or if T is a reference type then let `Deref_From` be `From` and `Deref_T` be `std::remove_reference_t<T>`:
    - If `Deref_T` is derived from `Deref_From` and if not during constant evaluation ([expr.const]), then the expression is profile-rejected by profile `std::type`.
    - Otherwise, if `Deref_T` is unrelated to `Deref_From`, then the expression is profile-rejected by profile `std::type`.

If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The `static_cast` operator shall not cast away constness ([expr.const.cast]).

4.4 `dynamic_cast` (Type.1.3, Type.1.4)

From [Pro.Type]

Type.1: Avoid casts

1.3: Don't cast between pointer types where the source type and the target type are the same

1.4: Don't cast between pointer types when the conversion could be implicit

This may be a style guideline and vary in different instantiations of a template or be to a dependent type that is sometimes the same or sometimes derived, so is deferred from the initial set of core profiles.

4.5 `(c_style)cast` (Type.4)

From [Pro.Type]

> Type.4: Don't use C-style (T)expression or functional T(expression) casts:
> Prefer construction or named casts or T{expression}

If this cast is a `const_cast`, `static_cast`, and/or `reinterpret_cast`, and the wording already says "The same semantic restrictions and behaviors apply," so this is covered by the previous wording.

Actually rejecting C-style and functional casts that are not in the above cases are style guidance and so deferred from the initial set of core profiles.

4.6 `function_style(cast)` (Type.4)

From [Pro.Type]

> Type.4: Don't use C-style (T)expression or functional T(expression) casts:
> Prefer construction or named casts or T{expression}

This cast is expressed in terms of cast expression and is "equivalent," so this too is covered by the previous wording.

4.7 Variable initialization (Type.5, Type.6)

From [Pro.Type]

> Type.5: Don't use a variable before it has been initialized: always initialize
>
> Type.6: Always initialize a data member: always initialize,
> possibly using default constructors or default member initializers

Always actually initialize an object – no vacuous initialization, no "no initialization" default-initialization.

Change [class.base.init] paragraph 9.3 as follows:

- otherwise, the entity is default-initialized ([dcl.init]); if default-initialization performs no initialization or initialization to an erroneous value, the expression is profile-rejected by profile `std::type` ([dcl.attr.profile]).

Change [basic.life] paragraph 2.0 as follows (leaving 2.1 onward the same):

> The *lifetime* of an object or reference is a runtime property of the object or reference. A variable is said to have *vacuous initialization* if it is default-initialized, no other initialization is performed, and, if it is of class type or a (possibly multidimensional) array thereof, a trivial constructor of that class type is selected for the default-initialization; if a variable has vacuous initialization or initialization to an erroneous value, its definition is profile-rejected by profile `std::type` ([dcl.attr.profile]). The lifetime of an object of type T begins when:

4.8 `va_arg` (Type.8)

From [Pro.Type]

Type.8: Avoid varargs: Don't use va_arg arguments

After [cstdarg.syn] paragraph 1, add the following:

If va_arg is used, the use is profile-rejected by profile `std::type` ([dcl.attr.profile]).

4.9 `union` (Type.7)

From [Pro.Type]

Type.7: Avoid naked union: Use variant instead

4.9.1 Option 1: Reject uses of unions

In [class.union.general] paragraph 5, add the following:

For a member access expression ([expr.ref]) that nominates a union member `M`, if

- the use of M is not the left operand of an assignment operator, and
- `M` is not part of the union's common initial sequence,

then the expression is profile-rejected by profile `std::type` ([dcl.attr.profile]).

4.9.2 Option 2: Check uses of unions (best-effort, does not require whole-program recompilation)

In [class.union.general] paragraph 5, add the following:

For an object U of union type, let U_{index} denote the index of U's active member. For a union with members $M_{0..N}$ in their lexically declared order, the member M_i has index `i`. Let *invalid* be the state of no member being active, and *unknown* be the state of an unspecified member being active. When the `std::type` profile is enabled, for each object U of union type that is not declared at namespace scope and is not a static data member:

- When U is created uninitialized, U_{index} is set to *invalid*.
- When the lifetime of `U.`M_i begins, U_{index} is set to `i`.
- When the lifetime of `U.`M_i ends, U_{index} is set to *invalid*.
- For a member access expression ([expr.ref]) that nominates a union member `U.`M_i, that is not part of U's common initial sequence, the expression `(`U_{index} `== i ||` U_{index} `==` *unknown*`)` is profile-checked by profile `std::type` ([dcl.attr.profile]).

- For every other use of U that would not be valid or would change meaning if U was const, U_{index} is set to *unknown*. *[Note:* For example, using U as an argument to a non-inline function that takes U by reference to non-const and might or might not modify it. *– end note]*

Notes Conceptually, one possible 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).

See [Sutter 2025] for a sample implementation that is wait-free (not just lock-free) for most operations. In my testing, even under heavy contention and oversubscription (3x more threads than CPU cores, doing no other work than >100 million union accesses (construction, set/get, destruction), with 10,000 union objects actively used at a time), the average overhead per union access was 6.2 CPU clock cycles (Clang), 11.7 cycles (GCC), or 15 cycles (MSVC).

To opt out of checking that the active member is correct, such as for a `union` access in a hot loop known to be intentional bit-swizzling or safe because of external tagging, the user can still use `[[profiles::suppress(std::type)]]` to eliminate any overhead.

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. (3) A `variant` is not a complete replacement for a union because of type and member anonymity; see [Sutter 2025] footnote.

5 **`bounds`** profile

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

5.1 Pointer arithmetic (Bounds.1)

From [Pro.Bounds]

> Bounds.1: Don't use pointer arithmetic. Use span instead

In [expr.add] paragraph 1, add the following:

> If the type of either operand is a pointer, the expression is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

In [expr.sub] paragraph 2, add the following:

> If the type of either expression is a pointer, the expression is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

In [expr.pre.incr] paragraph 1, add the following:

> If the type of the operand is a pointer, the expression is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

In [expr.post.incr] paragraph 1, add the following:

> If the type of the operand is a pointer, the expression is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

In [expr.pre.ass] paragraph 1, add the following:

> If the type of the left operand is a pointer, the expression is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

5.2 Array-to-pointer decay (Bounds.3)

From [Pro.Bounds]

> Bounds.3: No array-to-pointer decay

Modify [conv.array] paragraph 1 as follows:

> An lvalue or rvalue of type "array of `N` `T`" or "array of unknown bound of `T`" can be converted to a prvalue of type "pointer to `T`". The temporary materialization conversion ([conv.rval]) is applied. The result is a pointer to the first element of the array. An expression that perform this conversion is profile-rejected by profile `std::bounds` ([dcl.attr.profile]).

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's name cannot be silently passed to bounds-unsafe code that could attempt to perform arithmetic.

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

From [Pro.Bounds]

Bounds.4: Don't use standard-library functions and types that are not bounds-checked

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::as_const(a)+2` is a valid expression (see Notes below), then:
  - The standard-provided `std::index_in_range(a,b)` function is enabled, and is available without including any headers and checks `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
  - C perform a bounds check as-if a `contract_assert(std::index_in_range(a,b))` with `detection_mode::bounds`

In [utility], add a new subclause as follows:

x.x.10 index_in_range [utility.index_in_range]

```
constexpr bool index_in_range(auto const& a, auto const& b) noexcept;
```

Constraints: `as_const(a)[2]` is a valid expression and evaluates to `true`, and either:

- the type of `b` is signed integral and `std::ssize(a)` is a valid expression, or
- the type of `b` is unsigned integral and `std::size(a)` is a valid expression

Effects: Returns `0 <= b && b < max_size`, where `max_size` is either `std::ssize(a)` or `std::size(a)` depending if `b` is signed or unsigned respectively.

In [expr.sub] paragraph 2, add the following:

If the type of E2 is integral, then:

- if `requires(index_in_range(a,b))` is true, then the condition `index_in_range(a,b)` is profile-checked by profile `std::bounds`
- otherwise, if `requires(std::index_in_range(a,b))` is true, then the condition `std::index_in_range(a,b)` is profile-checked by profile `std::bounds`

Notes This enables opt-out of all checking for all objects of a container type by declaring:

```
void index_in_range(mytype const&, auto&&) = delete;
```

in the namespace of `mytype` or as a specialization of `std::index_in_range`.

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`" 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.

It used to also be unthinkable to bounds-check C++ programs. But times have changed: See the Google Security Foundations report [Rebert 2024] that showed adding bounds checking to entire C++ programs only cost 0.3% [sic] on modern compilers and optimizers. That is a fairly recent development and welcome surprise, as [Carruth 2024] elaborates.

Requiring `std::as_const(a)[2]` covers the cases of not only contiguous containers/views but also containers like `flat_set` and C++26 SIMD types. For test cases showing which containers correctly do and don't get bounds checks, see Frank Birbacher's test cases at https://godbolt.org/z/ocz7oPbEa.

6 `lifetime` profile

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

> **Note** All of these profiles are a start to build upon; especially this section is a start of a `lifetime` profile. The bulk of [Pro.Lifetime] is the static analysis in P1179, which per [P3465R1] SG23 direction in Wrocław (November 2024) is being pursued as a TS or whitepaper first, and as it succeeds can then be merged into `std::lifetime` in the IS.

6.1 Manual memory management (Lifetime.1)

In [expr.delete] paragraph 1, add the following:

> The expression is profile-rejected by profile `std::lifetime` ([dcl.attr.profile]).

In [c.malloc] paragraph 6, add the following:

> Invoking `free` is profile-rejected by profile `std::lifetime` ([dcl.attr.profile]).

6.2 Null dereference (Lifetime.1)

For every valid expression of the form `*p` where `p` is of type `P`:

- if `p!=P{}` is valid and boolean, then:
  - The standard-provided `std::indirection_not_null(p)` function is enabled, and is available without including any headers and checks `p!=P{}`
  - C perform a null check as-if a `contract_assert(std::indirection_not_null(p))` with `detection_mode::lifetime`

In [utility], add a new subclause as follows:

> **x.x.10 indirection_not_null [utility.indirection_not_null]**
>
> `template<typename P> constexpr bool indirection_not_null(P const& p) noexcept;`
>
> *Constraints:* `*p` and `p!=P{}` are valid expressions.
>
> *Effects:* Returns p != P{}.

In [expr.unary.op], change paragraph 1 as follows:

> The unary `*` operator performs *indirection*. Its operand shall be a prvalue of type “pointer to `T`”, where `T` is an object or function type. The operator yields an lvalue of type `T`. If the operand points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in [expr.typeid]. For an object p of a built-in or user-defined type:
>
> - if `requires(indirection_not_null(p)` is true, then the condition `indirection_not_null(p)` is profile-checked by profile `std::lifetime`

- otherwise, if `requires(std::indirection_not_null(p)` is `true`, then the condition `std::indirection_not_null(p)` is profile-checked by profile `std::lifetime`

[Note 1: Indirection through a pointer to an incomplete type (other than *cv* `void`) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see [conv.lval]. — *end note]*

Notes This enables opt-out of all checking for all objects of an iterator type that has a meaningfully dereferenceable default-constructed state by declaring:

```
void indirection_not_null(mytype const&) = delete;
```

in the namespace of `mytype` or as a specialization of `std::indirection_not_null`.

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

The compiler could choose to not emit this check (and not perform optimizations that benefit from the check) if it statically determines that `p` must be non-null at this source location (e.g., in an `if(p)` branch), or when targeting platforms that already trap null dereferences (e.g., platforms that mark low memory pages as unaddressable).

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`-enforced mode without any need for a library upgrade.

7 `arithmetic` profile

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

7.1 Narrowing conversions ([P3038R0] §12)

In [dcl.init.list] paragraph 7, add the following:

> For a narrowing conversion from type `From` to type `T`, if `T` is not `void` and `T` is not `bool`, the conversion is profile-rejected by profile `std::arithmetic` ([dcl.attr.profile]). Recommended practice: Implementations should emit a diagnostic that suggests using `std::narrow<T>(v)` ([utility.narrow]).

7.2 Signedness conversions ([P3038R0] §12)

In [conv.integral] paragraph 4, add the following:

> An integral conversion from a signed integer type to an unsigned integer type, or from an unsigned integer type to a signed integer type, is profile-rejected by profile `std::arithmetic` ([dcl.attr.profile]). Recommended practice: Implementations should emit a diagnostic that suggests using `std::narrow<T>(v)` ([utility.narrow]).

7.3 Arithmetic overflow ([P3038R0] §12)

Change [expr.pre] paragraph 4 as follows:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined and is profile-rejected by profile `std::arithmetic` ([dcl.attr.profile]).

8 Related papers / extension feedback

8.1 P3589R1

[P3589R1] argues that in addition to this syntax in prior profiles papers:

```
[[profiles::suppress(std::type)]]
```

it is also desirable to add two extensions.

(1) The ability to add a "reason" to document the rationale for suppressing a profile much as we have done already in the standard (e.g., with `static_cast`). For example, [P3589R1] proposes:

```
[[profiles::suppress(std::type, justification: "reason")]]
```

(2) The ability to suppress a specific rule, not just an entire profile. For example, [P3589R1] proposes something like this (using the IS section tag as a label; otherwise a label name would need to be introduced for each profile rule):

```
[[profiles::suppress(std::type, rule: "expr.reinterpret.cast"]]
```

(3) And doing both. For example:

```
[[profiles::suppress(
      std::type,
      rule: "expr.reinterpret.cast",
      justification: "reason"
      )
)]]
```

And:

- Using profile-designator instead of profile-name in the first sentence of "x.x.13 Profile attributes [dcl.attr.profile]" (see section 3).
- Defining the meaning of the `rule:` and `justification:` profile arguments.
- Optionally, adding a label to each profile rule (if we don't want to use just the IS section tag).

8.2 P3471R2, P3608R0, and P3611R0

[P3471R2] proposes dynamic checks that overlap with those in this paper. And as [P3471R2] Section 8.2 notes:

> *If profiles become a part of the Standard in the future, hardening can most likely be formulated as an additional profile…*

This can be implemented as a minor diff to the [P3471R2] wording, as:

> Additionally, an implementation can be a *hardened implementation*. A hardened implementation is one in which violating a *hardened precondition* is ~~a contract violation~~ profile-checked by profile `std::stdlib_hardened`.

[P3608R0] and [P3611R0] suggest adopting this, and deferring this paper's dynamic checks to post-C++26.

9 References

[Carruth 2024] C. Carruth. "Story-time: C++, bounds checking, performance, and compilers" (Blog post, November 2024).

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

[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.

[JSF++] Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle C++ coding standards for the system development and demonstration program" (Lockheed Nartin, December 2005).

[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).

[P3100R1] T. Doumler, G. Ažman, J. Berne. "Undefined and erroneous behaviour is a contract violation" (WG21 paper, October 2024).

[P3274R0] B. Stroustrup. "A framework for profiles development" (WG21 paper, May 2024).

[P3404R0] A. Kostur. "std::at: Range-checked accesses to arbitrary containers" (WG21 paper, September 2024).

[P3465R1] H. Sutter. "Pursue P1179 as a Lifetime Safety TS / whitepaper" (WG21 paper, December 2024).

[P3471R2] K. Varlamov, L. Dionne. "Standard library hardening" (WG21 paper, December 2024).

[P3589R0] G. Dos Reis. "C++ Profiles: The Framework" (WG21 paper, January 2025).

[P3589R1] G. Dos Reis. "C++ Profiles: The Framework" (WG21 paper, February 2025).

[P3608R0] V. Voutilainen. "Contracts and profiles: what we can reasonably ship in C++26" (WG21 paper, January 2025).

[P3611R0] B. Stroustrup. "Dealing with pointer errors: Separating static and dynamic checking" (WG21 paper, February 2025).

[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.

[Rebert 2024] A. Rebert, M. Shavrick, and K. Yasuda. "Retrofitting spatial safety to hundreds of millions of lines of C++" (Google, November 2024).

[Sutter 2025] H. Sutter. "My little New Year's Week project (and maybe one for you?)" (Blog post, January 2025).