Effect Sets: A Composable, Generalized `noexcept` for C++

Document number	P4312R0
Date	2026-07-15
Project	Programming Language C++
Audience	SG14 (Low Latency), EWG; CC: SG21 (Contracts), SG23 (Safety & Security)
Reply-to	Michael Galuszka <galuszka.michael@googlemail.com>

Abstract

C++ today carries exactly **one** aspect of a function's dynamic behaviour in its type: `noexcept` — the guarantee not to throw. This paper generalizes that principle into a small, closed, orthogonal **family of effects** (for example "does not allocate", "does not block", "does not recurse") that are declared as part of the function type, checked by the compiler, propagated through the call graph, and discharged at region boundaries. The proposal is **entirely static and decidable**, and therefore standardizable independently of any research-grade, quantitative resource-bound extensions. It follows a model already implemented in Clang (`[[clang::nonblocking]]` / `[[clang::nonallocating]]`) and the established treatment of `noexcept` as part of the type (since C++17).

Beyond the core design, two supporting analyses are integrated as first-class sections: the **form argument** (why effects must be a type specifier rather than an attribute, §4) and the **trust boundary and precise soundness statement** (how foreign and un-annotated code is handled, and exactly what "sound but incomplete" means, §7–§8).

Table of contents

Revision history

- **R0** — Initial version.

1. Motivation

For hard real-time paths, latency-critical middleware, and freestanding code, the decisive interface question is not *"what does this function compute?"* but *"which*

dynamic effects does it exclude?" — allocation, blocking, recursion, throwing exceptions. Today this guarantee is documented informally or forbidden wholesale by coding standards, rather than carried as a checked promise to callers.

`noexcept` proves that C++ can carry such an effect cleanly in the type system: it is declarable, checkable, part of type identity, sound across indirect calls, and equipped with well-defined conversion rules. This paper takes `noexcept` as a blueprint and generalizes it. The practical need is real and already implemented: the real-time audio community has for years relied on Clang's `nonblocking` / `nonallocating` effects to check exactly these guarantees at callback boundaries.

The timing is deliberate. C++26 — whose technical work was completed at the March 2026 Croydon meeting — includes Contracts (P2900) and Reflection (P2996), the carriers that later, out-of-scope facilities build on (§9), and the committee's stated direction for C++29 places safety at the centre of the cycle [P5000R1]. SG14's priority list for C++29/32 [P4029R1] frames its constituency's constraints as zero-overhead abstraction, predictable latency, and deterministic execution — precisely the properties that today cannot be stated, checked, or composed at function boundaries. This paper supplies the missing static half of that story.

---

2. The idea in one sentence

> *Generalize `noexcept` from a single effect ("does not throw") to a small, closed, orthogonal, and composable family of effects that is part of the function type and is checked statically by the compiler.*

---

3. Prior art

- **`noexcept`** (C++11; part of the type since C++17). The immediate precedent: type identity, covariant conversion (a `noexcept` function is usable where a throwing one is expected, not the reverse), override rules, deduction for special member functions.
- **Clang function effects** — `[[clang::nonblocking]]`, `[[clang::nonallocating]]` and their negations, with propagation checking through the call graph. An implementation existence proof for precisely this paper. Notably, Clang applies these to **function types** and treats them as conceptually a superset of `noexcept`; the override-conflict and call-graph-propagation behaviour described below already exists there. A January 2026 libc++ RFC began annotating the standard library for `[[nonblocking]]`, confirming both the model's practicality and the trust-boundary treatment of §7 (externally-defined functions are conservatively assumed potentially-blocking).
- **P3271 "function usage types"** (Lippincott) — a committee-blessed (SG21, post-MVP/C++29-timeframe) proposal in the *same design space*. Function usage types are named types that are **not** the type of any individual function; they sit on function pointer/reference types and describe the generalized expectations indirect callers have. P3271 deliberately keeps the property **out of** the individual function's type to allow backwards-compatible contractualization of existing code — the opposite tradeoff from this paper, which binds the effect to the function type for a published, definition-site, everywhere-visible guarantee. The two are complementary rather than exclusive (see §4); a usage type could in principle be formulated over effect-bearing function types.
- **Ada** — `pragma Restrictions (No_Dynamic_Allocation, No_Recursion, …)`; Ravenscar/Jorvik as certified restriction profiles. Demonstrates the value of categorical behavioural guarantees in certification.
- **`must_not_suspend`** (a lint in other languages) — precedent for "suspension as an effect" with coroutines.
- **PMR / allocators** — the basis for distinguishing allocation from pre-reserved

storage versus dynamic allocation (see §5.6, discharge).

---

4. Why a type specifier and not an attribute

This is the first question EWG will ask, and there is a fresh, documented precedent that frames it: C++26 deliberately rejected attribute syntax for Contracts in favour of a genuine grammatical construct (`pre` / `post` as a *function contract specifier*). Bringing an effect facility as an attribute, without addressing that precedent, loses the conversation before it starts. The argument is made here, and made affirmatively.

4.1 What an attribute cannot do, by definition

An attribute carries two properties in C++ that disqualify it from expressing an effect guarantee.

Attributes are ignorable. The standard expressly permits an implementation to **ignore an unknown attribute without diagnostic** ([dcl.attr.grammar]). That is the very purpose of attributes: optional, non-semantic information whose omission does not change program behaviour. For an effect guarantee this is fatal. A guarantee that a conforming implementation may silently ignore is not a guarantee; it is a comment with square brackets. `noexcept` is deliberately **not** an attribute, but a specifier with effect on type identity and on program behaviour — the immediate precedent for this proposal.

Standard `[[attributes]]` are not part of the type, and therefore break across translation units. A property that is not part of type identity cannot be carried reliably across separate compilation: it does not survive the type-based interface (function pointer, vtable slot, template argument). That defeats precisely the soundness of indirect and virtual calls (§5.3–5.4) on which the whole system rests. (Clang's `[[clang::nonblocking]]` is a nonstandard *type* attribute — the exception that proves the rule; tellingly, even it has no effect on name mangling.)

This break is **not hypothetical.** It is documented for C++26 Contracts in P3835R0[1], for exactly the scenario this proposal guards against: when a contract is bound not to type/interface identity but to a per-translation-unit, non-type-bound semantic, the linker may select, for a given call, a variant of the function in which the check does **not** occur — for instance when the call is not inlined and a client-emitted, `ignore`-compiled version is chosen. The promise is then lost with no diagnostic anywhere. An ignorable, non-type-bound property has this hole by construction. A type specifier does not, because ODR and type identity force consistency.

4.2 The break, concretely

Suppose `nonallocating` were an attribute `[[rt::nonallocating]]` rather than a type specifier:

```
        [[rt::nonallocating]] void process(Event&);

        void on_tick(Event& e) [[rt::nonallocating]] {
          void (*p)(Event&) = &process;   // the attribute is NOT part of the pointer type
          p(e);                            // p has type void(*)(Event&) — no guarantee
remains
        }
```

The pointer `p` has type `void(*)(Event&)`. The attribute, not being part of type identity, has **vanished** at the conversion to a pointer. The call `p(e)` is therefore a call through an unguaranteed pointer, and the checker can no longer know the target was `nonallocating`. Either it rejects every such call (unusable) or it lets it through (unsound).

With the type specifier (§5.3):

```
        void process(Event&) effects(nonallocating);

        void on_tick(Event& e) effects(nonallocating) {
          void (*p)(Event&) effects(nonallocating) = &process;  // effect is part of the
pointer type
          p(e);                                                 // OK: p carries the
guarantee
        }
```

Here `effects(nonallocating)` is part of the pointer type. Covariant refinement (§5.3) admits `&process → p` and forbids the reverse. The call is sound because the type carries the guarantee. This is the entire reason effects must be a type specifier.

4.3 The anticipated counter: “then a non-ignorable attribute”

One might propose solving the problem with a **non-ignorable** (“must-understand”) attribute rather than a type specifier. That addresses ignorability (§4.1, first point) but **not** type identity (§4.1, second point): a non-ignorable attribute still lies outside type identity and still does not travel across function pointers and virtual calls. The break of §4.2 would persist. Type identity is the load-bearing property; non-ignorability is necessary but not sufficient.

4.3a The third option: P3271 “function usage types”

The form question is not binary. P3271 (function usage types) is a committee-blessed, C++29-timeframe proposal that escapes the attribute-vs-type-specifier dichotomy: the property is carried by a **named type on the pointer/reference** (so it is not ignorable, unlike an attribute) but is **not part of the individual function’s type** (unlike this paper’s specifier). It is engineered for the same indirect-call problem as §4.2, with a different goal — backwards-compatible contractualization, since existing functions keep their type. The honest distinction: P3271 prioritizes backward compatibility (property on the pointer); this paper prioritizes a **published, definition-site, everywhere-visible** guarantee of the function itself (property on the function type, like `noexcept`). The two are complementary — a usage type could be formulated over effect-bearing function types — and this paper’s claim is specifically about the published-guarantee use case, not a rejection of P3271.

4.4 The dividing line

The clean test that separates the genuinely-attribute cases from the specifier cases is: **does ignoring the property change correctness?**

Property	Ignoring changes correctness?	Hence form
Effects (this paper)	**yes** (guarantee breaks across indirect calls)	type specifier
Quantitative resource bounds (out of scope here)	yes (a numeric promise breaks)	contract assertion (P2900 carrier)
Trace / observability (out of scope here)	**no** (only less observability)	library directive / attribute

Only where ignorability is correctness-neutral is the attribute form admissible. Effects are not such a case.

---

5. Detailed design

5.1 The effect vocabulary (closed, orthogonal)

A small set of orthogonal base effects fixed by the standard. What is annotated is always the **absence** of an effect (as with `noexcept`); unannotated means “no guarantee” and is

the default, backward-compatible state.

Effect guarantee	Meaning
`nonthrowing`	Throws no exception (identical to `noexcept`, see §5.7)
`nonallocating`	No dynamic/global memory request on the path
`nonblocking`	Does not block the **calling thread** (see definition below)
`nonrecursing`	No (direct or indirect) recursion

Definition of `nonblocking` (normative). The function does not block the **calling thread**: it holds no waiting lock, performs no blocking system call, and is subject to no unbounded synchronization delay. The effect concerns only the waiting behaviour of the calling thread. It does **not** denote the concurrency-theoretic *progress guarantee* (lock-freedom / wait-freedom), which is a property of a concurrent algorithm over the progress of multiple threads and is explicitly out of scope. This definition coincides with Clang's existing meaning, so the established name is retained rather than renamed.

Reserved for later revisions, deliberately not in R0: `nonsuspending` (coroutines), `nonsyscalling` / `noio`. The vocabulary stays **closed** — no user-defined effects — to keep the type system and composition decidable and manageable.

5.1a Relationship to Clang's effect hierarchy

Clang's implementation treats `nonblocking` as the stronger constraint: a function declared `[[clang::nonblocking(true)]]` together with `[[clang::nonallocating(false)]]` is rejected, because on mainstream hosted platforms the global allocator may take locks, so "does not block" subsumes "does not allocate". (Clang's parallel coupling of both effects to `noexcept` is covered in §5.7.)

This paper keeps the base vocabulary **orthogonal** and treats that subsumption as a *platform fact*, not a *type-system axiom*, for three reasons. First, the implication is contingent: an allocator that is itself lock-free — or one drawing from pre-reserved storage with a non-falling-through upstream (§5.6) — makes "allocating yet nonblocking" a coherent and useful state; baking the implication into type identity would make type identity platform-dependent. Second, orthogonal base effects keep the conversion and override rules (§5.3–5.4) a pure subset lattice with no special cases. Third, nothing is lost in practice: the strong bundle is named once via `effectset rt_safe = nonthrowing, nonallocating, nonblocking, nonrecursing;`, and an implementation remains free to warn when `effects(nonblocking)` appears without `nonallocating` on a platform whose global allocator is known to block — a quality-of-implementation diagnostic, which is exactly where Clang's rule lives today.

5.2 Syntax

An *effect-specifier* as part of the function type:

```
void on_audio_block(Buffer& b) effects(nonblocking, nonallocating);
```

Named, reusable sets:

```
effectset rt_safe = nonthrowing, nonallocating, nonblocking, nonrecursing;

void on_timer_tick(Event& e) effects(rt_safe);
```

5.3 Effects are part of the type

`void(Buffer&) effects(nonblocking)` is a type **distinct** from `void(Buffer&)`. Conversions follow covariant refinement — exactly analogous to `noexcept` conversion:

```
        void (*pany)(Buffer&);                          // no guarantee
        void (*pnb)(Buffer&) effects(nonblocking);      // guaranteed nonblocking

        pany = pnb;   // OK: the stronger guarantee is safely "forgotten"
        pnb  = pany;  // ill-formed: a guarantee cannot be conjured from nothing
```

This makes **indirect calls** sound: within a `nonblocking` function, calls may be made only through pointers whose type carries `nonblocking`. A call through an unguaranteed pointer conservatively introduces the worst-case effect and is therefore disallowed there.

5.4 Override rules (Liskov for effects)

An override may **strengthen, but not weaken**, the guarantees of the base declaration:

```
        struct Base { virtual void tick() effects(nonblocking); };

        struct A : Base { void tick() effects(nonblocking, nonallocating) override; }; //
OK: stronger
        struct B : Base { void tick() effects(nonblocking) override; };                 //
OK: equal
        struct C : Base { void tick() override; };                                      //
ill-formed: weaker
```

A call through `Base*` relies on `Base`'s promise; an override with more guarantees is substitutable, one with fewer is not. Consistent with the `noexcept` override rules — and already implemented in Clang, where a weaker override produces an effect-conflict diagnostic on declaration merging.

5.5 Inference inside, declaration at the boundary

- Where the **definition is visible** (inline, templates), the effect set is **inferred** from the body — no annotation burden.
- At **public interfaces** (only the declaration is visible), the set is **declared** and checked by the compiler against the definition.
- For special member functions (destructors, move operations), a deduction analogous to `noexcept` deduction is provided.

5.6 Propagation and discharge at region boundaries

A function's effect set is the **union** of the effects of all syntactically possible callees — minus the effects **discharged** at a region boundary. R0 specifies two concrete discharge constructs; more are an extension point.

Discharge of `throwing` by complete catching:

```
        void f() effects(nonthrowing) {
          try { may_throw(); }
          catch (...) { handle(); }   // throwing is discharged here → f stays nonthrowing
        }
```

Discharge of `allocating` by pre-reserved storage. The effect `allocating` denotes *dynamic/global* allocation. A request through an allocator that draws from a fixed, pre-reserved buffer does not, in general, introduce the effect — **but only under a condition that a naive rule misses.** A `monotonic_buffer_resource` whose buffer is exhausted falls back to its *upstream* resource, which by default is the global allocator; a discharge that accepted any arena allocation would therefore be **unsound**. The precise rule:

> *Discharge of `allocating` by an arena holds only if the arena's upstream resource is itself `nonallocating` (for example `null_memory_resource`, which throws on exhaustion rather than allocating) and the exhaustion path is either handled locally*

> *or allowed to propagate as an exception (which is compatible with* `nonallocating`*, though not with* `nonthrowing`*). Otherwise* `allocating` *remains in the effect set.*

```
        // Sound: upstream cannot fall through to the global heap.
        void g(std::span<std::byte> storage) effects(nonallocating) {
          std::pmr::monotonic_buffer_resource arena{
              storage.data(), storage.size(), std::pmr::null_memory_resource() };
          std::pmr::vector<int> v{&arena};   // draws from arena; on exhaustion the null
upstream
                                             // makes the failure observable instead of
silently
                                             // allocating from the global heap → effect
discharged
          // …
        }
```

The discharge is thus bound to a provably non-falling-through resource.

5.7 Relationship to `noexcept`

`noexcept` **is** the effect `nonthrowing` — the same property, two spellings. R0 retains `noexcept` as the established spelling for the throwing effect and defines `effects(nonthrowing)` as equivalent. Conversion and override rules are unified under the general effect model, so that no second, parallel rule set arises. (Clang already emits a diagnostic when `nonblocking` / `nonallocating` appear without `noexcept`, reflecting the same superset relationship.)

5.8 Effect polymorphism over templates (design option)

Generic code should **inherit** the effects of its arguments rather than fixing them. Two paths:

1. **By inference** (baseline): with a visible definition, inferred per specialization. `std::for_each(es, f)` is `nonblocking` exactly when `f` is — automatically.
2. **By declared effect polymorphism** (extension, open question §11): a spelling such as `effects(like(F))` that takes the effects explicitly from a parameter type:

```
        template <class F>
        void for_each_rt(std::span<Event> es, F f) effects(like(F)) {
          for (auto& e : es) f(e);
        }
```

R0 requires only path 1. Path 2 touches effect variables and is marked as follow-up work (cf. effect polymorphism in Koka, “keyword/effect generics” in other languages).

5.9 Relationship to `constexpr` / `consteval`

Effects constrain **run-time behaviour**. The portion of a `constexpr` / `consteval` call executed during constant evaluation is exempt; the effect check concerns only code that can execute at run time.

---

6. Decidability

The effect check is decidable (it adds no undecidability of its own). For a given instantiation and the direct call graph the check is decidable; indirect and virtual calls are handled soundly because the effect is **part of the type** (§5.3–5.4). `nonrecursing` is decidable over the static call graph (acyclicity; indirect calls are only permitted through effect-carrying pointers). There is **no** halting-problem reduction as there is for quantitative bounds — which is the reason to advance this facility separately.

The propagation rules are compact and each is decidable:

- *Primitive operation* `op`: its latent effect set is the fixed assignment (e.g. `throw` → `{throwing}`, `operator new` → `{allocating}`).
- *Sequence* `s₁; s₂`: union of the two.
- *Direct call* of a **checked** function `g`: the declared bound `D(g)`.
- *Indirect call* through a pointer with declared bound `Dₚ`: `Dₚ`. Sound because conversions only **enlarge** `Dₚ` (§5.3).
- *Virtual call* through `Base&`: `D(Base::m)`. Sound because every override satisfies "override ⊆ base" (§5.4).
- *Discharge at a region boundary* `B`: the region's effects minus those `B` discharges (e.g. a catch-all removes `throwing`).

Enforcement. An effect violation is **ill-formed, diagnostic required** — a static property like a type error, not a run-time construct. For the gradual migration of existing codebases an implementation-defined **audit mode** is provided that reports violations as warnings rather than errors; the conforming default mode diagnoses hard (the expectation of certification regimes: no silently broken guarantee).

---

7. The trust boundary (foreign and un-annotated code)

An effect-checked function almost unavoidably calls code whose effects are **not visible**: `extern "C"` functions without a definition, separately compiled translation units without annotation, function pointers from C APIs, `dlopen`'d code, inline assembly, type-erased calls (`std::function`). Two naive answers are both untenable: forbid everything unknown (sound but unusable) or trust everything unknown (usable but unsound). This section closes that gap through an **explicit, local, auditable trust boundary** — and it is what makes the soundness statement of §8 precise rather than aspirational.

7.1 The trust taxonomy

How an effect guarantee *comes about* becomes a first-class, reflectable property with four levels:

Level	Origin	Soundness	Discharge status
checked	definition visible, compiler verifies body ⊆ declared bound	guaranteed	`proven`
asserted	annotation on a declaration **without** definition (FFI)	trusted, unverified	`assumed`
assumed	local `effects_assume` block around a call	trusted, unverified, local	`assumed` + site
unknown	no annotation, no definition	worst case (all effects)	—

A function at level `unknown` carries the worst-case bound; a call to it from within a guaranteed function is therefore **ill-formed** — the safe default. Trust must be expressed **explicitly**, through one of the two constructs below.

This is not hypothetical: the January 2026 libc++ effort to annotate the standard library for `[[nonblocking]]` reports exactly this behaviour — when the compiler cannot see a function's body (an externally-defined symbol such as the dylib instantiation of `std::sort`, or the internally-called hardening-failure logger), it must treat the function as potentially blocking. The trust levels below are the disciplined way to record where that worst-case assumption is overridden.

7.2 Assertion at the declaration (FFI header)

```
extern "C" void legacy_poll() effects(nonblocking);   // asserted, not checked
```

The compiler cannot inspect the C definition; the guarantee is an **assertion carried at the declaration site** by the programmer (per documentation/knowledge). It is recorded as an `assumed` guarantee — verifiable at run time or by external analysis, never silently reported as proven.

7.3 Local assumption (`effects_assume`)

```
void on_tick() effects(nonblocking, nonallocating) {
  effects_assume(nonblocking, nonallocating) {
    third_party_callback();   // I assert: this honours these effects
  }
}
```

`effects_assume(A) { s }` directs the checker to treat the effects named in `A` as **absent** from the latent set of `s`, regardless of what `s` calls. The construct is deliberately: **locally bounded** (one block/one call, no global switch); **named** with exactly the assumed effects (no blanket "trust everything"); **greppable and countable** (the only place where soundness is delegated to a human); and **coupled to future runtime monitoring** (every assumption is a candidate for a runtime check in an instrumented build — a `nonallocating` assumption can be checked by an allocation guard).

---

8. The precise soundness statement

The statement comes in two parts — this is the honest form of "sound but incomplete".

Theorem 1 (Checked soundness). In a program **without** trust constructs (no `asserted` declaration, no `effects_assume`), if a function `f` with declared bound `D(f)` passes the effect check, then **every** runtime execution of `f` exhibits only effects in `D(f)`. In particular, every checked guarantee (a `non-X` for `X ∉ D(f)`) is **never** violated.

Proof sketch. Structural induction over the propagation rules (§6). Base case: primitive operations have correctly assigned latent effects. Inductive step per rule: sequence/branch by union/maximum; direct call by the IH over `D(g)`; indirect call by conversion monotonicity (the bound only grows, never shrinks); virtual call by the override rule "override ⊆ base"; discharge by the correctness of each discharge rule (a complete catch provably catches all paths; the corrected arena discharge of §5.6 provably does not fall through). Each rule is sound with respect to the operational effect semantics. ∎ (sketch)

Theorem 2 (Trust localization). In a general program, **every** deviation of an execution from a declared effect set is attributable to a lexically enclosing trust construct (an `effects_assume` block) or an `asserted` declaration. The set of such constructs in a translation unit is **statically enumerable**.

In other words: the effect system is sound **modulo the audited trust boundary**. The trust sites are exactly the places where soundness is delegated to a human or to runtime, and they are syntactically explicit and fully enumerable. (This is structurally the same soundness story as Rust's `unsafe` or `assume` in verification.)

8.1 The trust surface as a feature, not a caveat

Theorem 2 yields a tool directly: the compiler can list **all** trust sites of a translation unit — the *effect trust surface*. For certification (DO-178C, ISO 26262) this is precisely the required artefact: a finite list of "here we trusted rather than proved", each one a review, test, and runtime-monitoring candidate. The incompleteness of the system is thereby not hidden but converted into a checkable inventory.

8.2 Edge cases

Case	Handling
Inline assembly	worst case, unless wrapped in `effects_assume`
C function pointer	type carries no effects → worst case; trust via `asserted` typedef or `effects_assume` at the call
`dlopen` / dynamic loading	inherently a trust site; `effects_assume` at the call point
Separately compiled TU without annotation	worst case; LTO may reconstruct the effects, otherwise an `asserted` declaration
Type erasure (`std::function`)	the effects of the erased target must be captured at construction → an effect-parameterized `std::function` type (open question) or an assertion

---

9. Interaction with C++26 and the standard library

C++26's technical work was completed at the March 2026 (Croydon) meeting and it is now in its DIS approval ballot; its Contracts (P2900) and Reflection (P2996) facilities are already being implemented (both are merged in GCC trunk today, with other implementations in progress). The carriers the later facilities build on are therefore materializing, not merely accepted. This paper targets **C++29**, whose stated focus is further memory safety and the evolution of Profiles in SG23 — the natural home for a complementary, publishing interface guarantee.

- **C++29 direction (P5000R1):** the Direction Group names safety as a focus of the C++29 cycle; effect sets contribute a fully static, decidable class of behavioural guarantees to that agenda.
- **Safety profiles (SG23; P4186R0, P3589R2, P3081R2):** at Brno (June 2026), SG23 resolved to produce a Profiles specification targeting C++29, to be merged into the IS if ready in time or otherwise published as a whitepaper. Profiles and effect sets are complementary, not competing. A profile is a region-scoped set of analysis rules subsetting the language ("code in this region shall not use construct X"); an effect set is a property of a *function type*, carried through pointers, virtual dispatch, and conversions, and therefore sound across indirect calls — which region-scoped rules cannot express (§4.2). Conversely, a profile is the natural place to *demand* effects ("within this profile, callback parameters shall be `effects(nonblocking)`"), making the two features mutually reinforcing (§11.10). Notably, the safety taxonomy underlying the profiles effort (P2687, restated in P3700R0) already names **timing** and **concurrency** errors as safety categories, while the profiles currently advanced (type, bounds, initialization, lifetime) do not address them; effect sets supply the language mechanism for exactly that gap.
- **`noexcept`:** unified as `nonthrowing` (§5.7).
- **Reflection (P2996, C++26):** allows reading an entity's declared effect set at compile time — the basis for generated documentation and tooling (a future facility, not required here).
- **Contracts (P2900, C++26):** the form decision of §4 follows the same path P2900 took (specifier, not attribute); future quantitative extensions can ride P2900's grammar, but the facility proposed here is purely a type-system one. Note the concrete interaction surfaced by the libc++ work: under hardening with the `observe` assertion semantic, an inserted check can pull an externally-defined function into the call graph and change a function's effect set (e.g. `std::span::operator[]` calling an external hardening-failure logger) — a real coupling between this facility and future quantitative extensions that the design must account for.

- **Contracts momentum (P3097R3):** contract support for virtual functions was adopted into draft C++29 at Brno, less than three months after C++26 was finalized — the contracts substrate that such future quantitative extensions would ride is being actively extended.
- **Hardened standard library:** a libc++ RFC (January 2026) has begun annotating externally-defined functions for `[[nonblocking]]` on a best-effort basis; notably, the maintainers are reluctant to guarantee non-blocking-ness as a *stable* property of an attribute — which is itself an argument for binding the guarantee to the type (where stability is the norm, as with `noexcept`). A phased, standardized library-annotation programme remains follow-up work.
- **Function usage types (P3271):** a parallel, committee-blessed C++29-timeframe proposal occupying the same space via a different mechanism (a usage type on pointer/reference types rather than the function type); complementary, see §3 and §4.
- **Freestanding:** the natural first area of use.
- **Coroutines:** `nonsuspending` is deliberately deferred (§5.1), but the model is designed to admit it additively.

---

10. Wording sketch (informal)

Note: R0 provides only an informal sketch. Normative wording against a specific working draft, with stable-name references and marked edits, is required before this can progress beyond design review.

- `effect-specifier` as part of the *function-type* (attachment analogous to [except.spec]). Its content is a sequence of standard effect identifiers or a name introduced via `effectset`.
- **Type identity & conversion.** Two function types are identical only with the same effect set; a function (pointer) with a superset of guarantees is convertible to one with a subset, not the reverse.
- **Override.** An override may only extend (strengthen) the base's effect set, never reduce it; a violation is ill-formed.
- **Definition vs. declaration.** If a declaration carries an `effect-specifier`, the definition must honour it (diagnostic required); if it is absent with a visible definition, the set is inferred.
- **Discharge.** [except.spec]-style rules, extended by the discharge constructs of §5.6 (catch-all; non-falling-through arena).
- **Trust constructs.** `effects_assume( effect-list ) compound-statement` treats the listed effects as absent from the block's latent set; an `effect-specifier` on a declaration without a visible definition is an *asserted* guarantee. Both are recorded for the trust surface (§8.1).
- `effectset`**.** A named, compile-time-resolved effect set; purely an abbreviation, with no type semantics of its own.

---

11. Open questions

1. **Effect polymorphism** (`effects(like(F))`, §5.8) — syntax and inference rules for effect variables; interaction with overload resolution.
2. **The exact list of discharge constructs** beyond `try`/`catch` and non-falling-through arenas (e.g. lock-free paths for `nonblocking`).
3. **Effect-parameterized** `std::function` — capturing target effects under type erasure (§8.2).
4. **Granularity of** `effects_assume` — single expression vs. block; interaction with exceptions leaving the block; partial assumptions.

5. **Library-annotation programme** — order and scope of `nonallocating` / `nonblocking` marking in the standard library; ABI implications.
6. **Mangling** — since effects are part of type identity, some mangling impact is expected (as with `noexcept` in function-pointer types); the open questions are the exact scheme and its ABI consequences.
7. **Diagnostic quality** — explainability of *why* an effect was introduced (the path through the call graph), analogous to good `constexpr` diagnostics.
8. **Trust-surface format** — a standardized, machine-readable inventory format.
9. **Interaction with LTO/whole-program** — inference across translation units.
10. **Profile integration** — surface syntax and semantics for a profile to *require* effect guarantees on declarations within its scope (coordination with SG23; see §9).
11. **Alignment with P3271** — whether `effects(...)` on a function type and a P3271 usage type on a pointer/reference type can share checking machinery and conversion rules (§4.3a).

---

12. Recommended next step

R0 deliberately requires only: the closed vocabulary of §5.1 with the calling-thread definition of `nonblocking`, effects as part of the type with conversion and override rules (§5.3–5.4), inference/declaration (§5.5), propagation with the two discharge constructs (§5.6), unification with `noexcept` (§5.7), and the explicit trust boundary with its two constructs (§7) that makes the soundness statement (§8) precise. Effect polymorphism (§5.8), the effect-parameterized `std::function`, and the library-annotation programme follow in a later revision.

---

13. References

- [P0012] Make exception specifications part of the type system (C++17).
- [P2900] Contracts for C++ — in C++26 (finalized at the March 2026 Croydon meeting); establishes specifier-not-attribute as the precedent for behavioural specifications; four evaluation semantics (ignore, observe, enforce, quick-enforce).
- [P2996] Reflection — in C++26 (finalized 2026).
- [P3271] Lippincott, *Function usage types* (a.k.a. *Function Types with Usage*) — committee-blessed for the post-MVP / C++29 timeframe; named types on function pointer/reference that carry indirect-caller expectations without changing the function's own type. Discussed as the third design option in §3–§4.
- [P3835] Spicer, Voutilainen, García, *Contracts make C++ less safe — full stop* — cited for the cross-TU consistency hole of non-type-bound semantics; its "component"/"less safe" framing is contested (see [P3846], [P3400], and the note in §4.1).
- [P3846] *C++26 Contract Assertions, Reasserted* — rebuttal to [P3835]; disputes the "component" framing while conceding the underlying linker mechanism.
- [P3400] Labels for contract-assertion evaluation semantics — a proposed in-code control mechanism; under discussion, not yet consensus.
- [P5000R1] Direction for ISO C++29 — names safety as a focus of the cycle.
- [P4029R1] The SG14 Priority List for C++29/32 — zero-overhead abstraction, predictable latency, and deterministic execution as the constituency's constraints.
- [P4186R0] Bindels, *A Plan For Profiles* — SG23 process plan; at Brno (June 2026) SG23 resolved to produce a Profiles specification targeting C++29.
- [P3589R2] Dos Reis, *C++ Profiles: The Framework* — activation/suppression mechanics.
- [P3081R2] Sutter, *Core safety profiles for C++26* — type, bounds, initialization, lifetime.

- [P3700R0] Bindels, *Making Safe C++ happen* — restates the [P2687] taxonomy in which timing and concurrency errors are safety categories.
- [P2687] Stroustrup, Dos Reis, *Design alternatives for type-and-resource safe C++* — the underlying safety taxonomy.
- [P3097R3] Contracts for C++: Virtual functions — adopted into draft C++29 at Brno (June 2026).
- Clang function effects — `nonblocking` / `nonallocating` (Function Effect Analysis); applied to function types, superset of `noexcept`, with call-graph propagation and override-conflict diagnostics. libc++ RFC (January 2026) on annotating the standard library for `[[nonblocking]]`.
- Ada Reference Manual — `pragma Restrictions`; Ravenscar/Jorvik.

Notes

1. P3835R0 is an advocacy paper (*Contracts make C++ less safe — full stop*) whose framing is contested: P3846R0 (*C++26 Contract Assertions, Reasserted*) rejects its notion of a program "component" as a checkable unit, since evaluation semantics apply at translation-unit granularity and the language defines no consistent "component". What is **not** disputed is the mechanism relied on here — P3846R0 concedes that a non-inlined call to a weak (e.g. `inline`) function compiled under differing semantics across translation units resolves to a definition the linker selects arbitrarily, so the applied semantic is unpredictable. A proposed in-code remedy for the contracts case (labels, P3400R1) remains under discussion without consensus. The argument in this section takes no side in that debate: binding the guarantee to **type identity** removes the per-translation-unit selectability of the semantic altogether, so the failure mode both sides acknowledge cannot arise for an effect specifier. ↩