Zero-Overhead Abstractions

Building Flexible Vector Math Libraries with C++20

Greg von Winckel

CppCon 2025

September 15, 2025

SAND2025-11344 C

Introduction

- **The Challenge**: Writing high-performance, generic numerical code in C++ has historically involved significant compromises.
- **What we will cover**:
  - **The Problem**: "How do I make a generic vector math library able to use any appropriate data type without *depending* on that data type's library?" (Why traditional approaches fall short.)
  - **The Solution**: Modern C++ features
    - Concepts
    - Customization Point Objects
    - `tag_invoke`
  - **Alternatives**: A brief look at other approaches.
  - **A Convenient Tool**: We will provide a single-header utility library that makes writing, using, testing, and refactoring the CPOs easy.

Why This Matters

The pervasive need for real vector computing:

- **AI/ML**: The core of modern AI (deep neural networks, matrix/vector multiplications) is driving massive growth in data center power consumption.
- **Scientific Computing**: ODE/PDEs, multiphysics, numerical optimization, engineering simulation
- **Graphics & Gaming**: Fundamental to 2D/3D rendering (transformations, lighting), AR/VR.
- **Signal and Image Processing**: Computer vision, telecommunications, medical imaging.

The Problem: A History of Compromise in C++

- **Performance vs. Abstraction**: C++ has always offered the promise of high performance, but achieving it often meant sacrificing abstraction.
  - **C-style**: Fast, but not type-safe and hard to maintain.
  - **Object-Oriented Programming**: Better abstraction, but often at the cost of performance due to virtual function calls and heap allocations.
- **CRTP**: A step in the right direction, but it's complex, verbose, and doesn't fully solve the problem of extensibility.
- **Expression Templates**: A powerful technique for optimizing mathematical expressions, but they are difficult to write and debug.
- **Other approaches**: Traits, function templates, policies, type erasure, `std::variant`

Motivation for this Work

The Rapid Optimization Library (ROL) — `https://github.com/sandialabs/rol`

- A unique C++ library for numerical optimization with special emphasis on
  - Simulation-based (e.g. PDE-constrained) optimization
  - Stochastic optimization
  - Optimal control
  - Optimal experimental design
- Uses inheritance everywhere: vectors, objectives, constraints, algorithms, etc.
- This has some advantages, e.g. no data copy needed for user types unlike other optimizers.
- It is also problematic for supporting arbitrary elementwise functions of vectors and certain architectures such as CUDA.

When Object Oriented is Good

- **Runtime Polymorphism**: When you need to be able to choose the type of an object at runtime, OOP *may* be the right tool for the job. It's best when you need frequent new types across module/team boundaries, especially when they can be processed by existing code without modification.
- **Clear Interfaces**: Virtual functions define clear interfaces between different parts of a program, but **DON'T** use OOP *only* because you need to define an interface.
- **Encapsulation**: OOP helps to hide implementation details and protect data from unwanted modifications.
- **SOLID Applicability**: The design needs are such that ensuring the "five guidelines" of OOP can be followed without too great of complexity.
- **Hot Take**: OOP is often unfairly maligned due to the legacy of its misuse. It is like complaining about a hammer that damaged all the nuts and screws. Use the *correct tool* for each job!

Question: How many have tried to implement something like `virtual` from scratch as an exercise?

What Kinds of Algorithms Do We Want?

Take the Conjugate Gradient as a motivating example. We would like to write one function that can be used with any sequential data container that contains "real numbers."

Require: $A \in \mathbb{R}^{n\times n}$, $A = A^\top$, $\sigma(A) \subset (0,\infty)$, $b, x_0 \in \mathbb{R}^n$
Ensure: $x \approx A^{-1}b$

$r_0 \leftarrow b - Ax_0$
$p_0 \leftarrow r_0$
$k \leftarrow 0$
while $\|r_k\| >$ tol **do**
  $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
  $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
  $k \leftarrow k+1$
end while

Depends only on: `inner_product`, `scale_in_place`, `add_in_place`, `dimension`, `clone`.

Essential Operations for Most Real-Vector Algorithms

We can reasonably expect to need to implement five distinct kinds of operations

Operation	Mathematical Representation	Function Name
Add one vector to another	$y \leftarrow y + x, \;\; x, y \in \mathbb{R}^n$	`add_in_place`
Multiply vector by scalar	$y \leftarrow \alpha y, \;\; y \in \mathbb{R}^n, \;\; \alpha \in \mathbb{R}$	`scale_in_place`
Inner/dot product	$(x \cdot y) \in \mathbb{R}, \;\; x, y \in \mathbb{R}^n$	`inner_product`
Determine vector size	$\dim(x) = n, \;\; x \in \mathbb{R}^n$	`dimension`
Create or access scratch vector	shape-compatible temporary	`clone`

Assume `clone` performs a deep copy of shape (data may be uninitialized).

We can easily think of other functions that would be useful, but this is enough to start implementing algorithms.

OOP Implementation

Let's consider how we might an abstract base class for our (mathematical) vector

```
template<typename RealT, typename IndexT=int>
class Vector {
public:
  Vector() = default;
  virtual ~Vector() = default;
  virtual void add_in_place( const Vector& ) = 0;
  virtual void scale_in_place( RealT ) = 0;
  virtual RealT inner_product( const Vector& ) = 0;
  virtual IndexT dimension() const = 0;
  /* clone? */
};
```

- Even with this simple 5-method requirement, we are off to a troubling start.
- There is no way the `clone` method is flexible enough for every type of "scratch" vector we might like to have

OOP Implementation

- In order to have runtime polymorphism, we essentially need `clone` to return a pointer to a `Vector`.
- So much for leveraging views, arenas, returning by value, etc.
- Let's *pretend* for the moment that this is not an outrageous restriction.

```
#include <memory>

template<typename RealT, typename IndexT=int>
class Vector {
public:
  Vector() = default;
  virtual ~Vector() = default;
  virtual void add_in_place( const Vector& ) = 0;
  virtual void scale_in_place( RealT ) = 0;
  virtual RealT inner_product( const Vector& ) const = 0;
  virtual IndexT dimension() const = 0;
  virtual std::unique_ptr<Vector> clone() const = 0;
};
```

Implementing CG for `Vector`

Let's gloss over the matrix-vector multiplication and suppose something appropriate exists. Our function might have this signature

```
template<typename RealT, typename IndexT>
void conjugate_gradient(       Vector<RealT,IndexT>& x,
                         const Matrix<RealT,IndexT>& A,
                         const Vector<RealT,IndexT>& b,
                               RealT                 tol,
                               IndexT                maxIter );
```

Implementing CG for `Vector`

The first line of our CG algorithm already leads to frustration. We have $r_0 \leftarrow b - Ax_0$. We can implement this as

```
auto r = b.clone();
auto tmp = x.clone();
A.apply(*tmp);
tmp->scale_in_place(-1);
r->add_in_place(*tmp);
```

- We have to keep track of which symbols are pointers and which are not.
- CG (and many other algorithms) often include expressions that are linear combinations of vectors.
- It would be convenient to have an AXPY operation such as in $x_{k+1} \leftarrow x_k + \alpha_k p_k$.

Adding an AXPY method

No problem! We'll just add another method to our base class.

```
template<typename RealT, typename IndexT=int>
class Vector {
public:
  Vector() = default;
  virtual ~Vector() = default;
  // 5 pure virtual methods
  virtual void axpy_in_place( RealT alpha, const Vector& x ) = 0;
};
```

It's just one more method derived classes must override... but it *can* be decomposed into the other 5 operations. Perhaps we shouldn't *require* it.

Adding an optional AXPY method

Instead of making pure virtual, we can add a default implementation.

```
template<typename RealT, typename IndexT=int>
class Vector {
public:
  Vector() = default;
  virtual ~Vector() = default;
  // 5 pure virtual methods
  virtual void axpy_in_place( RealT alpha, const Vector& x ) {
    auto tmp = x.clone();
    tmp->scale_in_place(alpha);
    add_in_place(*tmp);
  }
};
```

Derived classes *should* override this for efficiency, but we do not require it.

On the use of `clone` in an OOP design

- Because the `clone` method dynamically allocates memory via `std::unique_ptr`, we want to use it *sparingly*.
- We check the algorithm and see what the minimum number of "new" vectors needed are.
- If we reuse `tmp` for something else, it would be convenient to have an assignment operator.
- These distractions are getting in the way. You just want to **implement the algorithm**.
- CG is a **simple** algorithm. What if we needed arbitary nonlinear functions?
- The **real problem** is that every time you call the algorithm, for every function call *in* the algorithm, you are repeatedly asking a question whose answer never changes: "What kind of `Vector` are you?"
- We used OOP to define an interface. This is misuse.
- We want to write the algorithm so that it works for **every type** that implements the operations the algorithm uses.
- There has to be **a better way!**

Comparing Approaches

Without going into how each approach would be implemented, here's a high-level overview of the pros and cons of OOP compared with well-known alternatives

Feature / Axis	OOP (Virtual)	Traits Class	Free Function Overloads	CRTP
Performance	⚠️ Runtime dispatch	✅ Zero-cost	✅ Zero-cost	✅ Zero-cost
Extensibility	❌ Inheritance only	✅ Easy (specialization)	✅ Easy via overload	❌ Tied to base class
Interoperability (STL/foreign)	❌ Needs wrappers	✅ Yes	✅ Yes	❌ No
Dynamic Dispatch	✅ Supported	❌ None	❌ None	❌ None
Discoverability	✅ Central base class	✅ Central trait	❌ ADL is scattered	✅ Localized via base
Diagnostics & Tooling	✅ Good	❌ Poor without concepts	❌ Poor unless wrapped	⚠️ Error-prone
Composability	❌ Hard to mix ops	✅ Modular	✅ Modular	✅ OK in closed set
Maintenance/Scalability	❌ Inflexible	✅ Scales well	⚠️ Hard to organize	⚠️ Fragile base tie
User Ergonomics	✅ Simple to call	⚠️ Verbose per type	⚠️ Simple to write, but hard to call correctly	✅ Familiar syntax

C++20 Concepts: A Revolution in Generic Programming

- **What are they**: Concepts are named sets of requirements on a type.
- **How they work**: The compiler checks if a type satisfies a concept at compile time.
- **Why they are useful**:
  - **Improved Error Messages**: No more pages of cryptic error messages when a type doesn't meet the requirements of a template.
  - **Easier to Read and Write**: Concepts make template code self-documenting.
  - **More Flexible**: Concepts allow us to write more generic code that can work with a wider range of types.

Instead of writing a base class `Vector` that has a pure virtual `add_in_place` method, we define a **concept**

```
template<typename T>
concept add_in_place_c = requires( T& y, const T& x ) {
  { add_in_place(y,x) } -> std::same_as<void>;
};
```

Later: The conditions for a callable `add_in_place` and type `T` to satisfy the concept.

A better approach to `clone`

Ideally, we'd either like `clone` to return a smart pointer to its argument type or a type that is convertible to its argument type. Helper concept and functions:

```
template<typename T>
concept deref_c = requires( T&& x ) {
  { *std::forward<T>(x) }; // Has dereference operator
};

template<deref_c T>
auto deref_if_needed(T&& x) noexcept(noexcept(*std::forward<T>(x)))
-> decltype(*std::forward<T>(x)) {
  return *std::forward<T>(x);
}

template<typename T>
requires (!deref_c<T>)
auto deref_if_needed(T&& x) noexcept -> T&& {
  return std::forward<T>(x);
}
```

A better approach to `clone`

Now we can define a concept that will ensure that we can `clone` an object and by passing the return value through the (possibly no-op) `deref_if_needed`, we will have *something* we can use as an argument to the over vector operation functions we need.

```
template<typename T>
concept clone_c = requires (const T& x) {
  { deref_if_needed(clone(x)) } -> std::convertible_to<T>;
};
```

We can define a far more flexible `clone` function for our type than what inheritance allows. If our vector size is known at compile time, we could stack allocate a workspace (arena) that provides access to available temporary vectors. No need to dynamically allocate vectors in algorithms if that is critical for our application.

```
auto x_cl = clone(x); auto& p = deref_if_needed(x_cl);
```

Now we don't care whether `clone` returns a pointer. No further special handling needed.

Dimension Concept

A dimension function should have an integral return type

```
template<typename T>
concept dimension_c = requires( const T& x ) {
  { dimension(x) } -> std::integral;
};

template<dimension_c T>
using dimension_t = decltype(dimension(std::declval<T>()));
```

Inner Product Concept

For purposes of simplicity, let's define

```
template<typename T>
concept real_scalar_c = std::floating_point<T>;
```

However, we could make a real scalar concept that also works for custom types, including multiprecision and rational types.

```
template<typename T>
concept inner_product_c = requires( const T& x, const T& y ) {
  { inner_poduct(x,y) } -> real_scalar_c;
};

template<inner_product_c T>
using inner_product_t = decltype(inner_product(std::declval<T>(),
                                               std::declval<T>()));
```

Scale in place Concept

This concept needs two template parameters

```
template<typename T, typename S>
concept scale_in_place_c = requires( T& y, S alpha ) {
  { scale_in_place(y,alpha) } -> std::same_as<void>;
};
```

We can later add the requirement that whatever `S` is, it must be convertible to `inner_product_t<T>`.

Note if the size of a scalar is larger than 3x the size of a pointer, we might prefer pass by `const` reference here.

Real Vector Concept

- Putting it all together

```
template<typename T>
concept real_vector_c = add_in_place_c<T>  &&
                        clone_c<T>         &&
                        dimension_c<T>     &&
                        inner_product_c<T> &&
                        scale_in_place_c<T,inner_product_t<T>>;
```

- We can now use this as the criteria to determine the set of types for which an algorithm like CG is defined.

- *However*, before we use this concept, we should *first* understand what it is actually checking for.

Why Not Free Functions?

Free-function overloads seem simple, but at scale they cause problems:

- **Unpredictable lookup**: Overload sets depend on ADL and ordinary lookup. Small, unrelated changes (hidden friends, template args, using-decls) can change which function is called.
- **Name collisions**: Any visible `add_in_place` participates in resolution. Library "fallbacks" compete with user overloads, creating ambiguities or hijacking calls.
- **No single extension point**: Customizations are scattered across namespaces. There's no central, discoverable place to implement or find the operation for a type.
- **Poor fallbacks & diagnostics**: Forcing ADL (e.g., with a deleted template) yields inscrutable errors when nothing matches. Valid defaults are hard to provide without interfering.
- **Composability limits**: A function name isn't an object: you can't pass it around as state, attach policy, or layer behavior cleanly. Boilerplate repeats per operation.

Takeaway: Keep the call site stable and predictable. Use a CPO (an object) plus `tag_invoke` for a single, hygienic extension point.

Qualified vs Unqualified Names

Qualified (explicit path)

```
std::vector<int> v;
MyNamespace::func();
object.member();
ptr->member();
```

Compiler looks exactly where you specify

Unqualified (no path)

```
vector<int> v;   // using std
func();          // Where is it?
swap(a, b);      // ???
```

Compiler must search for the name

When unqualified, C++ uses TWO search strategies...

Two Search Strategies for Unqualified Names

Ordinary Lookup

"The Local Detective" (Inspector Lestrade)

When: ALWAYS goes first

Searches for: ANY name
(variables, functions, types, templates)

Strategy: Start local, expand outward
current → outer → namespace → global

ADL

"The Specialist Detective" (Sherlock Holmes)

When: Only for function calls
with typed arguments

Searches for: ONLY functions
(never variables or types)

Strategy: Check argument namespaces
(all at once, no order)

Two Search Strategies for Unqualified Names

Ordinary Lookup

"The Local Detective" (Inspector Lestrade)

The Catch:

Stops at first matching name
(even if it's the wrong type!)

`using` declarations work

Cannot be disabled

ADL

"The Specialist Detective" (Sherlock Holmes)

The Power:

Finds ALL matches
(builds overload set)

`using` declarations ignored

Disable with `(func)(args)`

Two Search Strategies for Unqualified Names

Ordinary Lookup

"The Local Detective" (Inspector Lestrade)

ADL

"The Specialist Detective" (Sherlock Holmes)

Key Insight for CPOs:

Objects are found by ordinary lookup (predictable)
Functions can be found by ADL (unpredictable)

CPOs are objects to avoid ADL!

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type

```
namespace math {
  template<typename T>
  struct Vector {
    T x, y;
  };

  void add_in_place(       Vector<T>& lhs,
                     const Vector<T>& rhs);
}

static_assert(add_in_place_c<math::Vector<double>>);
```

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type
- ✅ that is a friend

```
namespace physics {
  template<typename T>
  struct Force {
    T magnitude;

    template<typename U>
    friend void add_in_place(      Force<U>& lhs,
                             const Force<U>& rhs);
  };
}

static_assert(add_in_place_c<physics::Force<float>>);
```

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type
- ✅ that is a `friend`
- ✅ in namespace associated with template arguments

```
namespace ctrs {
  template<typename T>
  struct Vec {
    std::vector<T> data;
  };
}

namespace math {
  template<typename U>
  struct Point { U x, y; };

  // ADL will find when trying T=Point
  template<typename T>
  void add_in_place(      ctrs::Vec<T>& lhs,
                    const ctrs::Vec<T>& rhs)
}

static_assert(add_in_place_c<ctrs::Vec<math::Pt>>);
```

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type
- ✅ that is a `friend`
- ✅ in namespace associated with template arguments
- ❌ in an unrelated namespace

```
namespace graphics {
  template<typename T>
  struct Color {
    T red, green, blue;
  };
}

namespace util {  // Unrelated namespace
  template<typename T>
  void add_in_place(      graphics::Color<T>& lhs,
                    const graphics::Color<T>& rhs);
}

static_assert(!add_in_place_c<graphics::Color<int>>);
```

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type
- ✅ that is a `friend`
- ✅ in namespace associated with template arguments
- ❌ in an unrelated namespace
- ❌ in the global namespace when type is not

```
namespace audio {
  struct Sample {
    float amplitude;
  };
}

// Global scope - ADL won't find this
void add_in_place(      audio::Sample& lhs,
                  const audio::Sample& rhs);

static_assert(!add_in_place_c<audio::Sample>);
```

Understanding Argument Dependent Lookup (ADL)

ADL would find a function...

- ✅ in same namespace as type
- ✅ that is a `friend`
- ✅ in namespace associated with template arguments
- ❌ in an unrelated namespace
- ❌ in the global namespace when type is not
- ❌ accessible only through `using` declaration

```
namespace game {
  template<typename T>
  struct Player {
    T health;
  };
}

namespace ops {
  template<typename T>
  void add_in_place(      game::Player<T>& lhs,
                    const game::Player<T>& rhs);
}

void test_function() {

  using ops::add_in_place;  // This doesn't help ADL

  static_assert(!add_in_place_c<game::Player<int>>);
}
```

Understanding Argument Dependent Lookup (ADL)

Another thing that ADL will **not** find are *functors*.

```
struct add_in_place_ftor {
  template<typename T>
  constexpr void operator()( T& y, const T& x ) const;
};

inline constexpr add_in_place_ftor add_in_place;
```

This fact motivated “customization point objects”

Customization Point Objects

```
namespace rvf {
struct add_in_place_ftor {
  template<add_in_place_c T>
  constexpr void operator() ( T& y, const T& x ) const
  noexcept(noexcept(add_in_place(y,x))) {
    add_in_place(y,x);
  }
};

inline constexpr add_in_place_ftor add_in_place; // CPO
} // namespace rvf
```

Customization Point Objects

We can use slightly more compact syntax.

```
namespace rvf {
inline constexpr struct add_in_place_ftor {
  template<add_in_place_c T>
  constexpr void operator() ( T& y, const T& x ) const
  noexcept(noexcept(add_in_place(y,x))) {
    add_in_place(y,x);
  }
} add_in_place; // <- ADL won't try this
} // namespace rvf
```

Where does the compiler search for `add_in_place`?

ADL:

- The namespace that contains the definition of type `T`.
- Any namespace enclosing `T`'s namespace.
- If `T` is a class type, the namespaces of any base classes of `T`.
- If `T` is a template instantiation like `std::vector<int>`, the namespaces associated with the template arguments (so both `std` for `vector` and the global namespace for `int`).

Normal unqualified lookup:

- The `rvf` namespace where the call is made.
- The global namespace.

ISO/IEC 14882:2023 paragraph 2 of [over.match.best] states that if there is no unique function that is better than all other viable functions, then the call is ill-formed. The standard uses precise language here: "the call is ill-formed" means the compiler must issue a diagnostic and reject the program.

Traditional CPO (no `tag_invoke`): How it works

```
namespace rvf {
  inline constexpr struct add_in_place_ftor {
    template<class T>
    constexpr auto operator()(T& y, const T& x) const
    noexcept(noexcept(add_in_place(y, x)))
    -> decltype(add_in_place(y, x)) {
      // 1) Ordinary lookup ignores the object, considers functions only
      // 2) ADL adds candidates from namespaces associated with T
      return add_in_place(y, x); // unqualified call on purpose
    }
  } add_in_place; // CPO object
}
```

- Relies on ADL to find a free function named `add_in_place` for `T`.
- You cannot safely put a fallback `add_in_place` in `rvf` because it would be found by ordinary lookup.
- Common workaround is a deleted “poison-pill” template to force ADL, which hurts diagnostics.

Problems with traditional (non-`tag_invoke`) CPOs

- **Name collisions**: Any function named like the operation visible in `rvf` participates in overload resolution via ordinary lookup. This can hijack calls or create ambiguities.
- **No safe fallback**: You can't provide a default `add_in_place` in `rvf`. Common "poison-pill" tricks (a deleted template) intentionally produce hard-to-read errors when ADL fails.
- **Unpredictable ADL sets**: Hidden friends, using-declarations, and template argument associated namespaces all influence which functions are found. Small changes elsewhere can break lookups.
- **Boilerplate per operation**: Each CPO must hand-roll layering (member vs free vs fallback), noexcept and return-type plumbing, and constraints. Easy to get subtly wrong.
- **Diagnostics**: When overload resolution fails, errors point at the wrong place (deep in the CPO), not the missing/ill-formed customization.

CPO with tag_invoke

```
namespace rvf {
inline constexpr struct add_in_place_ftor {
  template<typename T>
  constexpr auto operator() ( T& y, const T& x ) const
  noexcept(noexcept(tag_invoke(*this,y,x))) ->
  decltype(tag_invoke(*this,y,x)) {
    return tag_invoke(*this,y,x);
  }
} add_in_place; // CPO
} // namespace rvf
```

What `tag_invoke` buys you

- **Namespace hygiene**: The operation name is a type (the CPO object). Only `tag_invoke` is searched by ADL, avoiding collisions with `add_in_place` in `rvf`.
- **Single extension point**: Users customize by defining `tag_invoke(tag, ...)` in their associated namespace; no need to match your operation's name.
- **Safe fallbacks**: Combine with `tag_fallback_invoke` (or a constrained fallback) to supply defaults without interfering with ADL.
- **Better constraints/diagnostics**: You constrain the CPO in terms of `tag_invoke`; failures point at missing customizations, not mysterious overload sets.
- **Same zero-cost dispatch**: Resolution still happens at compile time; No runtime overhead compared (at least when passing by reference).

Customizing with `tag_invoke` (example)

```
// In user's namespace associated with the type
namespace math {
  struct vec { /* ... */ };

  // Teach rvf::add_in_place how to handle math::vec
  inline void tag_invoke(rvf::add_in_place_ftor, vec& y, const vec& x) {
    // elementwise add y += x;
  }
}

// Then generic code just calls the CPO
template<class T>
void axpy(T& y, const T& x) {
  rvf::add_in_place(y, x); // finds tag_invoke via ADL
}
```

References on CPOs and `tag_invoke`

- Customization Point Design in C++11 and Beyond (https://ericniebler.com/page/2/)
- N4381 - Suggested Design for Customization Points
- P1292R0 - Customization Point Functions
- P1665R0 - Tag based customization points
- P1895R0 - `tag_invoke`: A general pattern for supporting customisable functions
- Why `tag_invoke` is not the solution I want | Barry's C++ Blog
- P2279R0 - We need a language mechanism for customization points
- P2547R0 - Language support for customisable functions

As of right now CPOs + `tag_invoke` appears to be the *best* functional solution within the standard, but *significant* drawbacks remain.

- **Boilerplate**: Large amount of "code plumbing" needed per CPO
- **Error Prone**: Easy to make mistakes implementing. Catastrophic/inscrutable compiler errors when overload resolution fails.

No language standard solution before C++29.

TInCuP: Tag Invoke + Customization Points

- The TInCuP library was created to mitigate the pain points of writing customization point objects with `tag_invoke`.
- Get it here: `https://github.com/sandialabs/TInCuP`
- Defines a CRTP base class that provides diagnostics and introspection for your CPO and an extensive traits class.
- Provides a Python script for generating the significant amount of boilerplate needed and another to verify existing CPOs adhere to specific format for automated refactoring as C++ evolves.
- Intended as a "bridge technology" until customizable functions are supported by the standard.

Using TInCuP - Simple Case

- **What you type**: The CPO is specified via JSON. A Python script generates the C++ code from jinja2 templates.

```
cpo-generator '{"cpo_name": "add_in_place", \
                "args": ["$V&: y", "const $V&: x"]}'
```

- **Generic type**: The $ symbol indicates `V` is a template parameter, otherwise it would be treated as a concrete type.

Recommendation: Treat the generated CPO code like a `Makefile` generated by CMake (don't modify it).

Using TInCuP - Simple Case

- **What you type**: The CPO is specified via JSON. A Python script generates the C++ code from jinja2 templates.

```
cpo-generator '{"cpo_name": "add_in_place", \
                "args": ["$V&: y", "const $V&: x"]}'
```

- **Generic type**: The $ symbol indicates `V` is a template parameter, otherwise it would be treated as a concrete type.

- **Alternatively**: If you use Vim (VS Code and CLion integrations also available)

```
CPO add_in_place '$V&:y' 'const $V&:x'
```

Recommendation: Treat the generated CPO code like a `Makefile` generated by CMake (don't modify it).

Using TInCuP - Simple Case

- **What you get**:

```
inline constexpr struct add_in_place_ftor final
  : tincup::cpo_base<add_in_place_ftor> {
  TINCUP_CPO_TAG("add_in_place")
  inline static constexpr bool is_variadic = false;
  using tincup::cpo_base<add_in_place_ftor>::operator();
  template<typename V>
  requires tincup::invocable_c<add_in_place_ftor, V&, const V&>
  constexpr auto operator()(V& y, const V& x) const
  noexcept(tincup::nothrow_invocable_c<add_in_place_ftor, V&, const V&>)
  -> tincup::invocable_t<add_in_place_ftor, V&, const V&> {
    return tag_invoke(*this, y, x);
  }
```

Using TInCuP - Simple Case

- **What you get**:

```
template<typename V>
concept add_in_place_invocable_c =
  tincup::invocable_c<add_in_place_ftor, V&, const V&>;

template<typename V>
concept add_in_place_nothrow_invocable_c =
  tincup::nothrow_invocable_c<add_in_place_ftor, V&, const V&>;

template<typename V>
using add_in_place_return_t =
  tincup::invocable_t<add_in_place_ftor, V&, const V&>;

template<typename V>
using add_in_place_traits =
```

Argument DSL

Token/Pattern	Meaning	Example input	Generated signature fragment
$T	Generic by value	"$T: x"	template<typename T>(T x)
$T&	Generic lvalue reference	"$T&: x"	template<typename T>(T& x)
$T&&	Forwarding reference	"$T&&: x"	template<typename T>(T&& x) (fwd)
$T...	Generic parameter pack (value)	"$T...: xs"	template<typename... T>(T... xs)
$T&...	Generic lvalue reference pack	"$T&...: xs"	template<typename... T>(T&... xs)
$T&&...	Forwarding reference pack	"$T&&...: xs"	template<typename... T>(T&&... xs) (fwd)
$const T	Const-qualified generic	"$const T: x"	template<typename T>(const T x)
$const T&	Const lvalue reference	"$const T&: x"	template<typename T>(const T& x)
$volatile T&	Volatile lvalue reference	"$volatile T&: x"	template<typename T>(volatile T& x)
Concrete	Concrete type (value/lvalue)	"int: n"	int n
Concrete	Concrete type (rvalue)	"std::string&&: s"	std::string&& s

No Preprocessor Black Magic

- **Note**: The preprocessor macro is only for adding metadata and as a grep-friendly token.

```
#define TINCUP_CPO_TAG(name_str) \
  static constexpr std::string_view name = name_str; \
  static constexpr std::string_view qualified_name() noexcept { \
    return "tincup::" name_str; \
  }
```

- **CRTP Base Class**: Where the “magic” is.

Inside the CRTP Base Class cpo_base

```
// CRTP base class for all CPOs
template<typename Derived>
struct cpo_base : public cpo_introspection<Derived>,
                  public cpo_diagnostics<Derived> {
  template<typename... Args>
  constexpr void operator()(Args&&... args) const {
    this->enhanced_fail(std::forward<Args>(args)...);
  }
}; // struct cpo_base
```

Inside the CRTP Base Class `cpo_base`

```
// CRTP base class for all CPOs
template<typename Derived>
struct cpo_base : public cpo_introspection<Derived>,
                  public cpo_diagnostics<Derived> {
  template<typename... Args>
  constexpr void operator()(Args&&... args) const {
    this->enhanced_fail(std::forward<Args>(args)...);
  }
}; // struct cpo_base
```

ADL note. Because each user-defined CPO type derives from `tincup::cpo_base<Derived>`, the derived CPO's *associated classes and namespaces* include those of its base class. As a result, an unqualified call to `tag_invoke` with the CPO object as the first argument considers overloads found in `namespace tincup` during argument-dependent lookup. This enables library-defined defaults and diagnostics implemented as `tag_invoke` overloads in `tincup` to be found without extra qualification.

Inside the CRTP Base Class cpo_base

```
// CRTP base class for all CPOs
template<typename Derived>
struct cpo_base : public cpo_introspection<Derived>,
                  public cpo_diagnostics<Derived> {
  template<typename... Args>
  constexpr void operator()(Args&&... args) const {
    this->enhanced_fail(std::forward<Args>(args)...);
  }
}; // struct cpo_base
```

Consequences.

- Library-provided `tag_invoke` overloads in `tincup` participate in overload resolution automatically; user-provided overloads in the argument types' namespaces are still found via their own associated namespaces.
- The candidate set for ADL includes `tincup` by construction; keep overloads constrained to avoid unintended matches, and avoid introducing unconstrained friends in unrelated namespaces.

Helpful Concepts and Type Aliases

```
template<typename Cp, typename...Args>
concept tag_invocable_c = requires ( const Cp& cpo, Args&&...args ) {
  { tag_invoke(cpo,std::forward<Args>(args)...) };
};

template<typename Cp, typename...Args>
concept invocable_c = tag_invocable_c<Cp, Args...>;

template<typename Cp, typename...Args>
concept nothrow_tag_invocable_c = requires ( const Cp& cpo, Args&&...args ) {
  { tag_invoke(cpo,std::forward<Args>(args)...) } noexcept;
};

template<typename Cp, typename...Args>
concept nothrow_invocable_c = nothrow_tag_invocable_c<Cp, Args...>;

template<typename Cp, typename...Args>
using tag_invocable_t = decltype(tag_invoke(std::declval<Cp>(),
                                            std::declval<Args>()...));
```

Utility Class cpo_introspection

```
template<typename Derived>
struct cpo_introspection {
  // Standard introspection using derived type
  template<typename...Args>
  static constexpr bool valid_arg_types = requires { tag_invoke(std::declval<Derived>(),
                                                                 std::declval<Args>()...);};

  template<typename...Args>
  static constexpr bool is_nothrow = requires { { tag_invoke(std::declval<Derived>(),
                                                              std::declval<Args>()...) } noexcept;};

  template<typename...Args>
  using return_type = decltype(tag_invoke(std::declval<Derived>(), std::declval<Args>()...));

  // Clean alias for return types - eliminates typename/template keywords in generated code
  template<typename...Args>
  using result_t = decltype(tag_invoke(std::declval<Derived>(), std::declval<Args>()...));

  template<template<class> typename Predicate, typename...Args>
  static constexpr bool valid_return_type = Predicate<return_type<Args...>>::value;
};
```

Utility Class `cpo_diagnostics`

Improves compile-time error messages by checking for several common mistakes when a `tag_invoke` overload is not found. Instead of a generic error, it provides a specific hint by checking if the call would have been valid under one of the following conditions:

- **Pointer Dereferencing**: Detects if arguments are pointers or smart pointers that should have been dereferenced (e.g., using cpo(*ptr) instead of cpo(ptr)).
- **Const-Correctness**: Detects when const objects are passed to a CPO that expects a mutable (non-const) argument.
- **Argument Order**: For binary operations, it checks if the call would work by swapping the two arguments.
- **Arity**: Catches common mistakes in the number of arguments provided, such as calling a unary CPO with two arguments or vice-versa.
- **Combined Issues**: It can also detect when a combination of the above errors is present (e.g., a const pointer needs to be dereferenced and passed as non-const).
- **None Detected**: Produces a standard fallback error but still displays the full list of argument types to aid in debugging.
- **Toggle Diagnostics**: Selectively enabled or disabled with compiler definitions.

Compiler Explorer Example (godbolt.org/z/z8WqsYshj)

```cpp
template <class T>
struct Vector {
    T x, y;
};

inline constexpr struct normalize_ftor final
  : tincup::cpo_base<normalize_ftor> {
  TINCUP_CPO_TAG("normalize")
  using tincup::cpo_base<normalize_ftor>::operator();
  inline static constexpr bool is_variadic = false;
  template<typename V>
  requires tincup::invocable_c<normalize_ftor, const V&>
  constexpr auto operator()(const V& vec) const
  noexcept(tincup::nothrow_invocable_c<normalize_ftor, const V&>)
  -> tincup::invocable_t<normalize_ftor, const V&> {
    return tag_invoke(*this, vec);
  }
} normalize;

template<typename T>
Vector<T> tag_invoke( normalize_ftor, const Vector<T>& vec ) {
  auto norm = std::sqrt(vec.x*vec.x + vec.y*vec.y);
  return {vec.x/norm,vec.y/norm};
}

int main() {
  /* expected_error: CPO: No valid tag_invoke overload for CPO,
     but there IS a valid overload for the dereferenced arguments.
     Some arguments appear to be pointers/smart_ptrs that may need
     explicit dereferencing. Consider: cpo(*ptr) instead of cpo(ptr) */
  auto ptr = std::make_unique<Vector<double>>();
  normalize(ptr);  // expect enhanced diagnostics (dereference hint)

  const Vector<double> v{4, 8};
  auto n = normalize(v);  // OK
}
```

```
/app/raw.githubusercontent.com/sandialabs/TInCuP/main/single_include/tincup.hpp:760:19: error:
static assertion failed due to requirement 'always_false_v<std::unique_ptr<Vector<double>,
std::default_delete<Vector<double>>> &>': ARGUMENT TYPES: Inspect the template instantiation
above to see actual argument types
  760 |     static_assert(always_false_v<Args...>,
      |                   ^~~~~~~~~~~~~~~~~~~~~~~
/app/raw.githubusercontent.com/sandialabs/TInCuP/main/single_include/tincup.hpp:909:51: note: in
instantiation of template class
'tincup::cpo_diagnostics<normalize_ftor>::show_argument_types<std::unique_ptr<Vector<double>> &>'
requested here
  909 |     [[maybe_unused]] show_argument_types<Args...> display_types{};
      |                                                   ^
/app/raw.githubusercontent.com/sandialabs/TInCuP/main/single_include/tincup.hpp:1086:11: note: in
instantiation of function template specialization
'tincup::cpo_diagnostics<normalize_ftor>::enhanced_fail<std::unique_ptr<Vector<double>> &>'
requested here
 1086 |     this->enhanced_fail(std::forward<Args>(args)...);
      |           ^
<source>:54:12: note: in instantiation of function template specialization
'tincup::cpo_base<normalize_ftor>::operator()<std::unique_ptr<Vector<double>> &>' requested here
   54 |   normalize(ptr);  // expect enhanced diagnostics (dereference hint)
      |            ^
In file included from <source>:12:
/app/raw.githubusercontent.com/sandialabs/TInCuP/main/single_include/tincup.hpp:910:19: error:
static assertion failed due to requirement 'always_false_v<normalize_ftor>': CPO: No valid
tag_invoke overload for CPO, but there IS a valid overload for the dereferenced arguments. Some
arguments appear to be pointers/smart_ptrs that may need explicit dereferencing. Consider:
cpo(*ptr) instead of cpo(ptr)
```

More TInCuP Advanced Usage

By supplying string options, the TInCuP CPO generator will create both runtime and compile time call method overloads for each

```
{
  "cpo_name": "add_in_place",
  "args": ["$T&: y","$const T&:x"],
  "runtime_dispatch": {
     "type": "string",
     "dispatch_arg": "exec_policy",
     "options": ["sequenced",
                 "parallel"]
  }
}
```

More TInCuP Advanced Usage

```cpp
inline constexpr struct add_in_place_ftor final
  : tincup::cpo_base<add_in_place_ftor> {
  TINCUP_CPO_TAG("add_in_place")
  inline static constexpr bool is_variadic = false;
  using tincup::cpo_base<add_in_place_ftor>::operator();
  static constexpr struct sequenced_tag {} sequenced;
  static constexpr struct parallel_tag {} parallel;
  static constexpr struct not_found_tag {} not_found;

  inline static constexpr auto options_array
    = tincup::string_view_array<2>{"sequenced","parallel"};
```

More TInCuP Advanced Usage

```
template<typename T>
requires tincup::invocable_c<add_in_place_ftor, T&, const T&,
                             add_in_place_ftor::not_found_tag>
constexpr auto operator()(T& y, const T& x, std::string_view exec_policy) const
noexcept(tincup::nothrow_invocable_c<add_in_place_ftor, T&, const T&,
                                     add_in_place_ftor::not_found_tag>) {
  // Runtime dispatch for string
  tincup::StringDispatch<2> dispatcher(exec_policy, options_array);
  return dispatcher.receive([&](auto dispatch_constant) {
    if constexpr (dispatch_constant.value < 2) {
      if constexpr (dispatch_constant.value == 0) {
        return tag_invoke(*this, y, x, sequenced);
      }
      if constexpr (dispatch_constant.value == 1) {
        return tag_invoke(*this, y, x, parallel);
      }
    } else {
      return tag_invoke(*this, y, x, not_found);
    }
  });
}
```

More TInCuP Advanced Usage

```
  // Compile-time dispatch overloads for string
  template<typename T>
  requires tincup::invocable_c<add_in_place_ftor, T&, const T&, sequenced_tag>
  constexpr auto operator()(T& y, const T& x, sequenced_tag) const
  noexcept(tincup::nothrow_invocable_c<add_in_place_ftor, T&, const T&, sequenced_tag>) {
    return tag_invoke(*this, y, x, sequenced);
  }
  template<typename T>
  requires tincup::invocable_c<add_in_place_ftor, T&, const T&, parallel_tag>
  constexpr auto operator()(T& y, const T& x, parallel_tag) const
  noexcept(tincup::nothrow_invocable_c<add_in_place_ftor, T&, const T&, parallel_tag>) {
    return tag_invoke(*this, y, x, parallel);
  }
  template<typename T>
  requires tincup::invocable_c<add_in_place_ftor, T&, const T&, not_found_tag>
  constexpr auto operator()(T& y, const T& x, not_found_tag) const
  noexcept(tincup::nothrow_invocable_c<add_in_place_ftor, T&, const T&, not_found_tag>) {
    return tag_invoke(*this, y, x, not_found);
  }
} add_in_place;
```

Real Vector Framework

`https://github.com/sandialabs/RealVectorFramework`

- **Depends on**: TInCuP to define its CPOs.
- **Core operations**: `add_in_place`, `clone`, `dimension`, `inner_product`, `scale_in_place`.
- **Advanced ops**: `axpy_in_place`, `binary_in_place`, `l2norm`, `unary_in_place`, `variadic_in_place`, `ReLU`, `softmax`.
- **Memory**: Observers and tools for creating and using memory arenas.
- **Algorithms**: CG, L-BFGS, gradient descent with bound constraints, truncated CG trust region, plus a few simple transformer models.

Support for `std::ranges::range`

If you use an STL container that satisfies the `range` concept, you do not need to write *any* `tag_invoke` functions.

```
template<std::ranges::range R>
void tag_invoke( add_in_place_ftor, R& y, const R& x ) {
  std::ranges::transform(y, x, std::ranges::begin(y), std::plus<>{});
}

template<std::ranges::range R>
  requires std::copy_constructible<R>
auto tag_invoke( clone_ftor, const R& x ) {
  return R(x);
}

template<std::ranges::range R>
auto tag_invoke( dimension_ftor, const R& r ) {
  return std::ranges::size(r);
}
```

Support for `std::ranges::range`

If you use an STL container that satisfies the `range` concept, you do not need to write *any* `tag_invoke` functions.

```
template<std::ranges::range R>
auto tag_invoke( inner_product_ftor, const R& x, const R& y ) {
  using value_type = std::ranges::range_value_t<R>;
  return std::inner_product(std::ranges::cbegin(x),
                            std::ranges::cend(x),
                            std::ranges::begin(y),
                            static_cast<value_type>(0));
}

template<std::ranges::range R>
void tag_invoke( scale_in_place_ftor, R& y,
                 std::ranges::range_value_t<R> alpha) {
  std::ranges::for_each(y, [alpha](auto& ye){ ye *= alpha; });
}
```

rvf::conjugate_gradient

```
template<typename Matrix, real_vector_c Vec>
requires self_map_c<Matrix, Vec>
void conjugate_gradient( const Matrix& A,
                         const Vec& b,
                         Vec& x,
                         vector_value_t<Vec> relTol = 1e-5,
                         vector_value_t<Vec> absTol = 0,
                         vector_size_t<Vec> maxIter = 100 ) {

  auto tol = rvf::fmax(relTol * rvf::l2norm(b), absTol);
  auto b_cl = rvf::clone(b); auto& r = rvf::deref_if_needed(b_cl);
```

rvf::conjugate_gradient

```
A(r, x);
rvf::scale_in_place(r, -1.0);
rvf::add_in_place(r, b);

auto rho0 = inner_product(r, r);
if(rvf::sqrt(rho0) < tol) return;

auto r_cl = rvf::clone(r); auto& p  = rvf::deref_if_needed(r_cl);
auto x_cl = rvf::clone(x); auto& Ap = rvf::deref_if_needed(x_cl);
```

rvf::conjugate_gradient

```
  for(vector_size_t<Vec> iter = 0; iter < maxIter; ++iter) {
    A(Ap, p);
    auto pAp = rvf::inner_product(Ap, p);
    auto alpha = rho0 / pAp;
    rvf::axpy_in_place(x,  alpha,  p);
    rvf::axpy_in_place(r, -alpha, Ap);
    auto rho = rvf::inner_product(r, r);
    if(rvf::sqrt(rho) < tol) break;
    auto beta = rho / rho0;
    rvf::scale_in_place(p, beta);
    rvf::add_in_place(p, r);
    rho0 = rho;
  }
}
```

Arbitrary Unary Functions $x \leftarrow f(x)$

Vim:
```
CPO unary_in_place '$V&:target' '$F&&:func'
```

Shell:
```
cpo-generator '{"cpo_name": "unary_in_place", \
                "args": ["$V&: x", "$F&&: func"]}'
```

```
inline constexpr struct unary_in_place_ftor final
  : tincup::cpo_base<unary_in_place_ftor> {
  TINCUP_CPO_TAG("unary_in_place")
  inline static constexpr bool is_variadic = false;
  template<typename F, typename V>
  requires tincup::invocable_c<unary_in_place_ftor, V&, F>
  constexpr auto operator()(V& x, F&& func) const
  noexcept(tincup::nothrow_invocable_c<unary_in_place_ftor, V&, F>)
  -> tincup::invocable_t<unary_in_place_ftor, V&, F> {
    return tag_invoke(*this, x, std::forward<F>(func));
  }
} unary_in_place;
```

Arbitrary Unary Functions $x \leftarrow f(x)$

```
template<std::ranges::range R, typename F>
requires unary_in_place_invocable<F, std::ranges::range_value_t<R>>
void tag_invoke( unary_in_place_ftor, R& y, F&& func ) {
  std::ranges::for_each(y, [func = std::forward<F>(func)](auto& ye) mutable {
    ye = func(ye);
  });
}
```

Arbitrary Binary Functions $x \leftarrow f(x, y)$

Vim:
```
CPO binary_in_place '$V&:x' '$F&&:func' 'const $V:y'
```

Shell:
```
cpo-generator '{"cpo_name": "binary_in_place", \
               "args": ["$V&: x", "$F&&: func", "const $V&:y"]}'
```

```
inline constexpr struct binary_in_place_ftor final
  : tincup::cpo_base<binary_in_place_ftor> {
  TINCUP_CPO_TAG("binary_in_place")
  inline static constexpr bool is_variadic = false;
  template<typename F, typename V>
  requires tincup::invocable_c<binary_in_place_ftor, V&, F, const V>
  constexpr auto operator()(V& x, F&& func, const V y) const
    noexcept(tincup::nothrow_invocable_c<binary_in_place_ftor, V&, F, const V>)
    -> tincup::invocable_t<binary_in_place_ftor, V&, F, const V> {
    return tag_invoke(*this, x, std::forward<F>(func), y);
  }
} binary_in_place;
```

Arbitrary Binary Functions $x \leftarrow f(x,y)$

```
template<std::ranges::range R, typename F>
requires binary_in_place_invocable<F, std::ranges::range_value_t<R>>
void tag_invoke( binary_in_place_ftor, R& x, F&& func, const R& y ) {
  std::ranges::transform(y, x, std::ranges::begin(y), std::forward<F>(func));
}
```

Arbitrary Variadic Functions $x \leftarrow f(x, y, z, \ldots)$

Vim:
```
CPO variadic_in_place '$V&:x' '$F&&:func' 'const $Vs&...:args'
```

Shell:
```
cpo-generator '{"cpo_name": "variadic_in_place", \
               "args": ["$V&: x", "$F&&: func", "const $Args&...:args"]}'
```

```
inline constexpr struct variadic_in_place_ftor final
  : tincup::cpo_base<variadic_in_place_ftor> {
  TINCUP_CPO_TAG("variadic_in_place")
  inline static constexpr bool is_variadic = true;
  template<typename F, typename V, typename... Vs>
  requires tincup::invocable_c<variadic_in_place_ftor, V&, F, const Vs&...>
  constexpr auto operator()(V& x, F&& func, const Vs&... args) const
  noexcept(tincup::nothrow_invocable_c<variadic_in_place_ftor, V&, F, const Vs&...>)
    -> tincup::invocable_t<variadic_in_place_ftor, V&, F, const Vs&...> {
  return tag_invoke(*this, x, std::forward<F>(func), args...);
  }
} variadic_in_place;
```

Arbitrary Variadic Functions $x \leftarrow f(x, y, z, \dots)$

```
template<typename F, typename T, typename...Args>
concept variadic_in_place_invocable =
  std::convertible_to<std::invoke_result_t<F,T,Args...>,T> &&
  (std::is_same_v<T,Args> && ...);

template<class OutputIt, class F, class FirstInput, class...RestInput>
OutputIt variadic_transform( OutputIt     out_begin,
                             OutputIt     out_end,
                             F&&          func,
                             FirstInput   first,
                             RestInput... rest ) {
  while(out_begin != out_end) {
    *out_begin++ = std::forward<F>(func)(it_inc(first), it_inc(rest)...);
  }
  return out_begin;
}
```

Arbitrary Variadic Functions $x \leftarrow f(x, y, z, \dots)$

```
template<std::ranges::range R, typename F, typename...Args>
requires variadic_in_place_invocable<F, std::ranges::range_value_t<R>,
                                        std::ranges::range_value_t<Args>...>
void tag_invoke( variadic_in_place_ftor, R& x, F&& func, const Args&... args ) {
  return variadic_transform(
    std::ranges::begin(x),
    std::ranges::end(x),
    std::forward<F>(func),
    std::ranges::begin(args)...
  );
}
```

Gradient Descent with Bound Constraints

```
// Objective function concept
template<typename F, typename Vec>
concept objective_function_c = requires(const F& f, const Vec& x, Vec& grad) {
  { f.value(x) } -> std::convertible_to<vector_value_t<Vec>>;
  { f.gradient(grad, x) } -> std::same_as<void>;
};

// Bound constraints representation
template<real_vector_c Vec>
struct bound_constraints {
  Vec lower, upper;

  // Project x onto [lower, upper] bounds
  void project(Vec& x) const {
    binary_in_place(x, [](auto xi, auto li) { return rvf::fmax(xi, li); }, lower)
    binary_in_place(x, [](auto xi, auto ui) { return rvf::fmin(xi, ui); }, upper)
  }
};
```

Future Work

- **RealVectorFramework**:
  - Implement execution policies
  - Add more algorithms
  - Recreate ROL with CPOs instead of inheritance
- **TInCuP**:
  - Review P2547R0 and P2279R0
  - Great ideas in these papers, but no implementation given yet
  - It should not be too difficult to write a clang extension that uses an approach like TInCuP under the hood
  - Distribute it and seek independent testing.
  - A compelling possibility for C++29
  - In the meantime, use TInCuP as a bridge technology with the promise of easy refactoring should a new language mechanism be introduced.

Questions?

http://github.com/sandialabs/TInCuP