woven BY TOYOTA

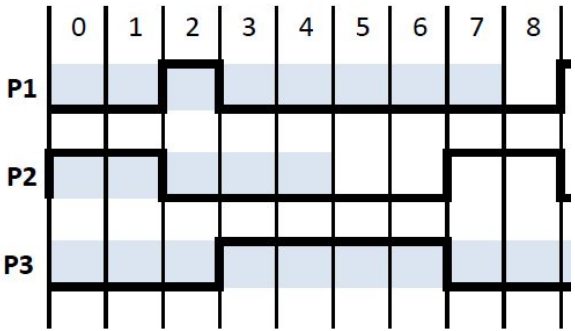
# Arene Base

## A Foundational C++ Library for Safety-Critical Software

13th May, 2026

**Charles Hussong**
Senior Runtime Framework Engineer
charles.hussong@woven.toyota

# Is this you?




Timing diagram from Wikipedia user Henryf

```
bazel_dep(
    name = "abseil-cpp",
    version = "20260107.0",
)
bazel_dep(
    name = "platforms",
    version = "0.0.11",
)
bazel_dep(
    name = "re2",
    version = "2025-11-05.bcr.1",
)
```

## Real-time safety

This class is just what I need, but it might allocate or throw an exception. I have an execution deadline, I can't afford that!

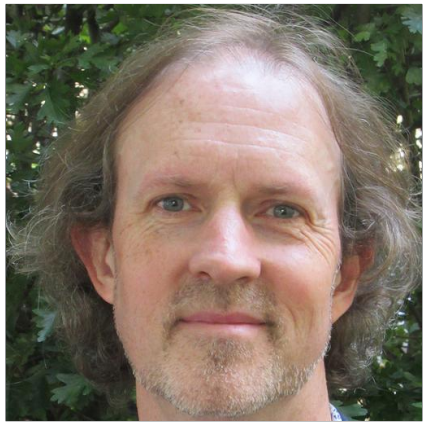
## Compiler/standards support

I want to use the real-time-safe parts of modern C++, but they're in C++20 or 23 and my compiler or coding standard only goes up to 17 (MISRA) or 14 (AUTOSAR)!

## Supply chain risk

I've cobbled together an environment that provides everything I need, but it depends on a bunch of generic open source libraries that will never be able to pass safety certification!

# Modern C++ helps code be both fast and safe. We should be able to use it on cars, where both are critical. Arene makes this possible.



Anthony Williams

Oliver Lee

Adam Panzica

Ryan Wooster

Charles Hussong

Luke Peterson

# Index



# 01. Standard Library

Features backported to C++14 from future standards;
partial implementation of the C++14 Standard Library

# Backported Standard Library Types

## Vocabulary Types from later C++, in C++14/17

Modern C++ has introduced a lot of Standard Library types to improve safety and expressiveness, but many of them are not available in C++14 or 17. We have backported several of the most important types, following the specification closely so they can be used as drop-in replacements for the future Standard ones. Undefined behavior and exceptions are generally replaced by hardened precondition checks.

## Provided in arene::base with full constexpr support:

- std::span
- std::string_view
- std::optional
- std::variant (including std::visit)
- std::expected (as arene::base::result)
- std::array

# Constexpr algorithms

Modern C++ allows us to "have our cake and eat it too" by offloading a lot of processing to compile-time. But many algorithms do not support this until C++17 or C++20. Arene Base provides the updated compile-time algorithms for C++14.

```cpp
struct MessageDef {
  unsigned priority;
  // other proprietary metadata
};

template <size_t Cap>
constexpr array<MessageDef, Cap>
priority_sort(span<MessageDef const, Cap> input) {
  array<MessageDef, Cap> arr{};
  arene::base::copy(
    input.begin(),
    input.end(),
    arr.begin()
  );
  arene::base::sort(
    arr.begin(),
    arr.end(),
    [](auto const& def1, auto const& def2) {
      return def1.can_id < def2.can_id;
    }
  );
  return arr;
}

constexpr auto prioritized_messages =
  priority_sort(generated_messages);
```

# Linear Algebra

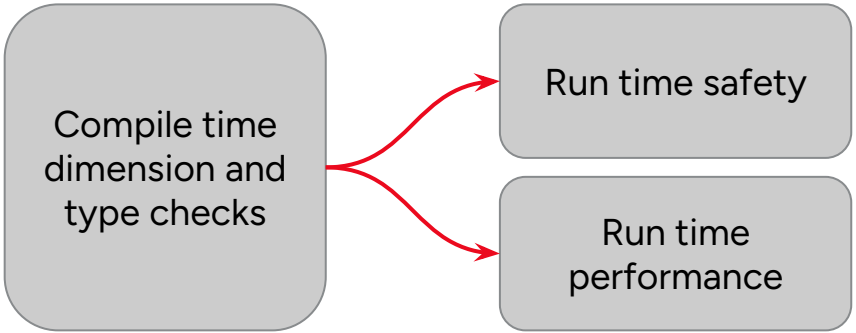
## Current Scope

The following are being implemented as backports of the [mdspan] and [linalg] sections of the C++23 and C++26 standards:

- Generic multi-dimensional "mdspan" views into 1-dimensional data
- Matrices and vectors with either compile- or run-time dimensions
- Basic matrix arithmetic (addition, multiplication, etc.)
- Some BLAS 1 algorithms (norms, transposition, etc.)
- Some BLAS 2/3/LAPACK algorithms (linear solve, LU factorization)
- Implementation uses std::ranges-like function objects so additional algorithms can be added relatively easily

## The Goal

We want enough performance and safety to process image data in real time locally on a safety-critical ECU. We get both by checking as much as possible at compile time, and making it easy for the user to provide the information needed to do so.



# Wait a minute...

OK, all this talk of backported future Standard Library features is great, but I don't have a *current* Standard Library in my environment. This a car after all, we don't have the luxury of using the latest compiler!

Good news: Arene Base provides a minimal implementation of the **C++14** Standard Library as well, to be used in environments where a compiler is provided without a Standard Library. Most features which do not require dynamic allocations, exceptions, or threading are included.

| More or less fully implemented (in some cases as backported version in arene::base namespace) | | | | | |
|---|---|---|---|---|---|
| <array> | <chrono> | <initializer_list> | <iterator> | <limits> | <ratio> |
| <tuple> | <type_traits> | <utility> | | | |
| **Most non-allocating parts implemented** | | | | | |
| <algorithm> | <exception> | <functional> | <memory> | <numeric> | <stdexcept> |
| **Currently not implemented** | | | | | |
| <bitset> | <complex> | <system_error> | <typeindex> | <typeinfo> | |
| **Will likely never be implemented** | | | | | |
| <codecvt> | <queue> | <random> | <stack> | <valarray> | |
| **Requires threading or dynamic allocation** | | | | | |
| <atomic> | <condition_variable> | <deque> | <forward_list> | <fstream> | <future> |
| <iomanip> | <ios> | <iosfwd> | <iostream> | <istream> | <list> |
| <locale> | <map> | <mutex> | <new> | <ostream> | <regex> |
| <scoped_allocator> | <set> | <sstream> | <streambuf> | <string> | <strstream> |
| <thread> | <unordered_map> | <unordered_set> | <vector> | | |

## Most embedded-relevant headers are available

We followed what was needed for Arene Base itself, but ended up implementing most things you might want to use on a safety-critical embedded system.

## 02. New Types and Algorithms

Non-allocating containers, type traits and mixins, physical units, etc.

# Inline containers

### Non-allocating fixed-capacity containers

All of these classes have a capacity set at compile time, but the size can vary dynamically within that capacity. They work with almost any element type (including non-trivial ones), and will never perform a dynamic allocation or throw an exception.

- inline_vector
- inline_deque
- inline_set
- inline_map
- inline_string
- inline_function

```
1  struct data {
2    // some data
3  };
4
5  void foo() {
6    constexpr std::size_t cap = 25;
7    inline_vector<data, cap> vec;
8
9    while (vec.size() < cap) {
10     vec.push_back(create_data());
11   }
12
13   while (!vec.empty()) {
14     for (auto& elem : vec) {
15       do_stuff(elem);
16     }
17     vec.pop_back();
18   }
19 }
```

# Type lists

When using or testing templates, it's often useful to create, manipulate, and iterate over lists of types. Arene Base provides a number of algorithms for manipulating type lists that work on any list template, e.g. GoogleTest's `Types`.

In almost all cases, these algorithms are implemented **non-recursively** so that compile times are significantly better than naive approaches.

```
// ----- Generated file -----

struct CanFrameDefMuteRadio {
  static constexpr CanId can_id;
  // Other proprietary metadata
};

struct CanFrameDefUnmuteRadio {
  static constexpr CanId can_id;
  // Other proprietary metadata
};

using CanFrameDefs = type_list<
  CanFrameDefMuteRadio,
  CanFrameDefUnmuteRadio>;

// ----- Non-generated file -----

using FdFrameDefs = type_lists::filter_t<
  CanFrameDefs, CanFrameMustBeFd>;
using SortedFdFrameDefs = type_lists::sort_t<
  FdFrameDefs, HasLowerCanId>;

using CanFdMonitor = GenericCanFdMonitor<
  SortedFdFrameDefs>;
```

# Mixins for your classes

Defining your own math type and want to make it fully comparable without repeating a lot of boilerplate? Just define a 3-way compare and use our comparison mixin.

Defining your own container but want to make sure it's usable with the standard algorithms? Use our mixins to easily define its iterators to be fully compliant with the standard concepts.

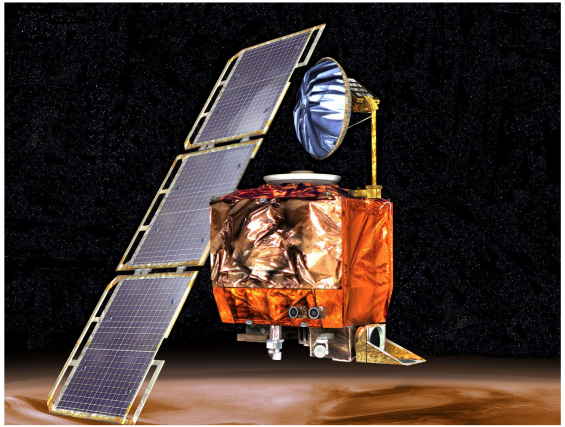
```
1  struct MyInt : ordering_from_compare<MyInt> {
2    int value;
3
4    static constexpr strong_ordering compare(
5        MyInt const& left, MyInt const& right) {
6      return compare_three_way{}(
7        left.value, right.value);
8    }
9  };
10
11 static_assert(MyInt{0} < MyInt{1});
12 static_assert(MyInt{2} >= MyInt{1});
13 static_assert(MyInt{1} == MyInt{1});
```

```cpp
// Watch out: Maneuver gives results in US units
void CraftState::update_expectations(
    Maneuver const& maneuver) {
  // If the storage types are compatible, simply
  // adding them will apply the conversion.
  expectation_.altitude =
    current_.altitude +
    maneuver.altitude_gain();

  // The conversion can be applied explicitly to
  // show that possible information loss is OK.
  expectation_.altitude =
    current_.altitude +
    maneuver.altitude_gain().in<meter>();

  // Quantities that are incompatible regardless
  // of unit can not be added, even with a cast.
  expectation_.altitude =                    // X
    current_.altitude +                      // X
    maneuver.radial_impulse().in<meter>();   // X
}
```

# Units and Quantities



With Arene Base's Units and Quantities library (WIP), you can perform compile-time validation and conversion of physical quantities with no runtime overhead.

SI units are built-in and support checking/conversion following the International System of Quantities. Additional units and quantities can be added by the user and optionally related to those of the SI/ISQ.

## 03. Testing and Safety

Full test coverage on everything; public features to help test user code more thoroughly

# Testing and Coverage

### Tested and Ready for Certification

All of Arene Base has extensive test coverage. In order to get this for the Standard Library, we implemented our own GoogleTest clone that doesn't use the Standard Library at all, so it runs anywhere, including a microcontroller without an OS.

The whole library was implemented following AUTOSAR C++14 and automatically produces artifacts needed for ISO 26262 certification at any ASIL level. If you need them, talk to us!

### Quality Standards

- Full AUTOSAR C++14 compliance verified by static analysis
- 100% line coverage
- 100% branch coverage
- 85% MC/DC coverage
- Death tests of all detectable precondition violations
- Tests that constexpr functions work at compile time

# Testing Your App

In addition to the test coverage of Arene Base itself, we provide tools to help test your own code.

Many of these are extensions to GoogleTest, which either build on official GoogleTest or our clone, and either allow testing extra properties like constexpr support or allow enabling/disabling tests based on template parameters.

A basic QuickCheck-style property-based testing library is currently under construction, which can assist with testing numerical algorithms.

```
TEST(MyTestSuite, MyTestCase) {
  constexpr MyType value{};

  // GoogleTest assertion; only checks at runtime
  ASSERT_EQ(value, value);

  // Built-in static assertion; no coverage
  static_assert(value == value, "");

  // Asserts at both compile- and runtime
  STATIC_ASSERT_EQ(value, value);
}

CONSTEXPR_TEST(MyTestSuite, MyConstexprTest) {
  // value doesn't need to be constexpr, but if
  // it isn't constexpr-usable this is an error.
  MyType value{};

  // Mutating value is also OK, but produces an
  // error if any step is not constexpr-usable.
  value += 10;

  // Asserts at both compile- and runtime
  CONSTEXPR_ASSERT_EQ(value, MyType{} + 10);
}
```

# Testing Our Classes

Some template classes in Arene Base have the tests templated as well, so you can instantiate and run them on the specific specializations that your app uses. This should give 100% coverage of all the functions that are available in your instantiation.

```
using MyTestTypes = ::testing::Types<
    inline_vector<std::int32_t, 25>,
    inline_vector<NonCopyableInt, 10>>;

// If the value type is not a built-in arithmetic
// type, this template must be specialized so
// Arene Base knows how to instantiate it.
template <>
NonCopyableInt arene::base::testing::test_value
<NonCopyableInt>(std::size_t idx) {
  constexpr array<int, 4> underlying{2, 3, 5, 8};
  return NonCopyableInt{underlying[idx % 4]};
}

// "MyApp" is the name of this instantiation of
// the InlineVector tests. It uses MyTestTypes.
ARENE_INSTANTIATE_TESTS(
  MyApp, InlineVector, MyTestTypes);

// All tests are instantiated and run for the
// int32 vector, while just the tests that don't
// require copying are instantiated and run for
// the non-copyable integer vector.
```

# Platform Support

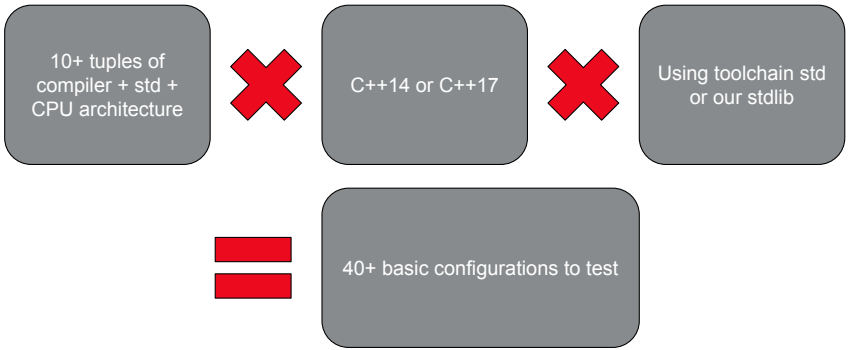
## Supported platform variation

Arene Base is built and tested in a wide variety of configurations, and the code attempts to detect and handle changes in what the specific compilers and environments support. With a few limitations in cases where it doesn't make sense, the whole library is tested with:

- C++14 and 17
- The toolchain-provided Standard Library and our own
- GCC and Clang
- x86-64 Linux, arm64 Linux and Darwin, and arm32 without an OS
- Exceptions enabled and disabled
- Dynamic sanitizers enabled and disabled

## Continuous integration

Arene Base's internal CI currently builds and tests the entire library with approximately 50 combinations of settings in order to cover the configuration space outlined at left. We use Bazel caching extensively to speed this up. The public repository has a simplified version of the CI.



Arene Base is available now on GitHub. Please let us know what you think!

# Thank you

github.com/woven-by-toyota/arene-base