Round 1: Containers, views and algorithms.
Memory handling and abstractions, linear algebra support.
What this presentation is or is not?
Motivation

The classic trio:
- Containers, iterators and algorithms.

Generalization and abstraction of the STL
- `std::ranges` and `std::concepts`

HPC use case: focus on contiguous memory.
- `std::vector<T,A>`, `std::array<T,N>`, `std::valarray<T>`...
- But, what about my super vector `V<...>` or `T*`?

Going further with views
- Can we catch the contiguous memory layout, unified the interface and enhanced the generated code?
std::span in C++20

A lightweight abstraction of a contiguous sequence of values of type `T` somewhere in memory.

A `std::span` can either have:

- **a static extent**, in which case the number of elements in the sequence is known at compile-time and encoded in the type,
- **a dynamic extent.**

It’s a non-owning view.

- Never allocate or deallocate.
- Handle raw pointer (no smart pointer handling)

**Why using it?**

- Adapt any chunk of unidimension contiguous memory.
- Unlock the STL power (ranges, algorithms, for range-based loops, ...).
- Span based code does not own the memory.
- Help the compiler with easier static analysis.
Let’s play with some code!
Multidimensional containers

What about multidimensional based code?
- Matrix
- Tensors

Classic approaches
- Using third party libraries: Eigen, blaze, xtensor...
- Write your own library
- Use low level primitives
Using third party libraries

Pros

○ High level interface
○ Collection of domain specific algorithms
○ EDSL / Expression Template oriented
○ Most of the time optimized by experts

Cons

○ Does it suit your needs?
○ You may need to extend its support for your application
○ Can be a closed and self contained environment
○ Does it provides standard compliant support?
Write your own container

Pros

- Full control
- Design will fit your needs
- You decide which level of abstraction you need

Cons

- Will your abstraction be composable with third party libraries or the standard?
- You have your hands on everything, maybe you're not an expert on everything?
- Does the support will last for your users?
Use low level primitives

**Pros**
- Full control
- Design will fit your needs

**Cons**
- You should really know what you are doing.
- No abstraction, verbosity
- Code readability is bad.
- No standard/third library compliance.
std::mdspan in C++23

A multidimensional std::span

○ Generalization of std::span (static, dynamic, lightweight, …)
○ Still a non-owning view

Why using it?

○ Adapt any chunk of multidimensional contiguous memory
○ High level abstraction on top of the memory
○ Unlock std::linalg support coming in C++26
○ Highly customisable for your needs
○ An adaptor to bridge the gap between third party libraries, low level memory handling and standard code.

Reference implementation available until full support
Free function linear algebra interface based on the BLAS.

- Standard BLAS calls in `cpp`
- Linking your preferred BLAS implementation under the hood
- `std::mdspan` as parameters
- Catching compile time dimension and sizing to generate optimized calls.
- Algorithms work with most of the matrix storage formats that the BLAS standard supports

Reference implementation available until full support
Let’s play with some code!
Let’s wrap it up!

Generic multidimensional code available in the standard

Design opportunities for the HPC community

Code composability made easier

Interaction with C++ idioms

Reference implementations available

Going further: standard parallelism!?
Any thoughts or questions?

To be continued...

...in round 2.