TAKING TEMPLATES ONE STEP FURTHER

WITH OPAQUES TYPES AND GENERIC NTTPS

May 7, 2021

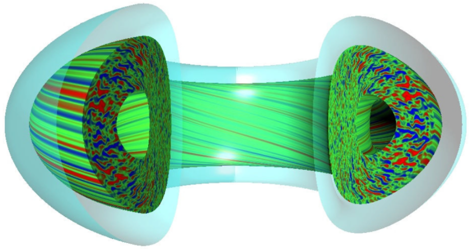
Joel FALCOU - Vincent REVERDY

Some Context

Why do we even array ?

Computations as a science pillar

- Simulations replaced experiments
- Fast computers are time machines
- Users are mainly scientists though



Enter the Matrix

- A nD-array must be **fast**
- A nD-array must be **easy to use**
- A nD-array must be **expressive**

How to design such a pervasive data structure ?

Challenges

A proper nD-array must be fast

- Must be usable with modern hardware (SIMD, GPGPU, ...)
- Abstractions should not hinder performances
- Must protect users from performance anti-patterns

A proper nD-array must be easy to use

- Must be intuitive for numeric-savvy users
- Must be customizable for power users

A proper nD-array must be expressive

- Numeric code should look numeric
- Combination of expressions should evaluate intuitively

Existing solutions

View/container

- `std::vector/std::array`
- `Boost.QVM`
- `std::span`
- `std::mdspan`

Expression-templates

- `Blitz++`
- `Eigen`
- NT^2
- `Armadillo`
- `Blaze`

Existing solutions

Why are those solutions not adequate ?

Concerns are to be separated

- Lazy evaluation
- nD-array handling
- Customization protocols
- Hardware support

Why are those solutions not adequate ?

Concerns are to be separated

- Lazy evaluation (C++Con 2019)
- nD-array handling
- Customization protocols
- Hardware support (C++Europe 2021)

Why are those solutions not adequate ?

Concerns are to be separated

- Lazy evaluation (C++Con 2019)
- nD-array handling
- Customization protocols
- Hardware support (C++Europe 2021)

SOLVE EACH ISSUE IN ITS OWN SOFTWARE COMPONENT

- Maximize re-usability
- No Monolith effect

Why are those solutions not adequate ?

Concerns are to be separated

- Lazy evaluation (C++Con 2019)
- nD-array handling
- Customization protocols
- Hardware support (C++Europe 2021)

SOLVE EACH ISSUE IN ITS OWN SOFTWARE COMPONENT

- Maximize re-usability
- No Monolith effect

Today we will care about the nD-array handling and customization issues by dissecting our nD container library : kiwaku

Why designing API is hard

Exploiting Compile Time Information

- Compilers need high-level information to enable high-quality optimization
- Users must be able to pass such information directly from the source

Why designing API is hard

Exploiting Compile Time Information

- Compilers need high-level information to enable high-quality optimization
- Users must be able to pass such information directly from the source

Source of Implementation Leaks

- Untyped values as template parameters
- Rigid template API that limits library's evolution and usability
- Improper compile-time/runtime separation of concern

Why designing API is hard

Exploiting Compile Time Information

- Compilers need high-level information to enable high-quality optimization
- Users must be able to pass such information directly from the source

Source of Implementation Leaks

- Untyped values as template parameters
- Rigid template API that limits library's evolution and usability
- Improper compile-time/runtime separation of concern

Examples

- `-1` as a dynamic size tag for `std::span`/`std::mdspan`
- `Eigen::Matrix<typename Scalar, int Rows, int Cols>`
- Passing allocator as type+value instead of pure value

Layout of the talks

Runtime components handling

- Full runtime components should be handled at runtime
- No need for type-based specification
- **Kiwaku solution: Opaque types**

Optimizations specifications:

- Array and view behavior options must be trivial to setup
- Users should have access to an intuitive option passing API
- **Kiwaku solution: Keyword parameters**

Compile-time/Runtime Barrier

- Compile-time options must have a rich semantic
- **Kiwaku solution: Non Type Template Parameters**

Tips #1 - Opaque Types

Kiwaku Allocator - What do we want

Kiwaku container constructors

```cpp
// Dynamic array using the default allocator
kwk::array<float,kwk::_2D> a1(kwk::of_shape(200,200))

// Dynamic array using some other allocator
kwk::array<float,kwk::_2D> a2(kwk::of_shape(10,50), some_allocator{});

// Allocator and data are copied to a1
a1 = a2;
```

Challenges

- How can we get rid of passing the allocator type as a template parameter ?
- Can we ensure proper copy and move semantic ?

Opaque Types

Definition

- A type is **opaque** if you can't see through it
- i.e the contents of its implementation is not accessible directly
- Such types are often implemented using **type-erasure**
- If users can't look at one type's internals, they are less opportunity for abstraction leaks

State of the Art

- `Based on Sean Parent's talk on Polymorphism`
- Use polymorphism as an implementation detail instead of as a first class property
- Provides a full Regular Type interface on top of the polymorphic behavior
- Does not require intrusive adaptation from user code

Opaque Types

Definition

- A type is **opaque** if you can't see through it
- i.e the contents of its implementation is not accessible directly
- Such types are often implemented using **type-erasure**
- If users can't look at one type's internals, they are less opportunity for abstraction leaks

Opaque type in the wild

- `FILE*`, the Great Old One
- `std::any`, `std::function` among others
- Louis Dionne's Dyno: `https://github.com/ldionne/dyno`

Opaque Types in API Design

Use Case: Dynamic Arrays

- Allocation of semi-large to really-large numeric arrays
- Allocations are often out of critical path
- Few resizing and growth (no `push_back`)
- Allocator can be more than just wrapping `malloc`/`free`

Our Setup

- Each allocator is written as an Alexandrescu's Allocator
  - Deals with block of `void*`
  - Knows about the size of the allocated block
  - Allocators can be chained/selected via arbitrary policies

Opaque Types in API Design

Use Case: Dynamic Arrays

- Allocation of semi-large to really-large numeric arrays
- Allocations are often out of critical path
- Few resizing and growth (no `push_back`)
- Allocator can be more than just wrapping `malloc`/`free`

Our Setup

- Each allocator is written as an Alexandrescu's Allocator
- Allocator definition on user side must be simple
    - No CRTP
    - No polymorphic base class

Opaque Types in API Design

Use Case: Dynamic Arrays

- Allocation of semi-large to really-large numeric arrays
- Allocations are often out of critical path
- Few resizing and growth (no `push_back`)
- Allocator can be more than just wrapping `malloc`/`free`

Our Setup

- Each allocator is written as an Alexandrescu's Allocator
- Allocator definition on user side must be simple
- Allocator must be `SemiRegularType`
  - No need to deals with complex traits
  - Allocators are just gonna be copied along their tables

Allocator design in Kiwaku

Basic Block of Memory

```cpp
struct block
{
  explicit operator bool() const { return length != 0; }

  friend bool operator!=(block const& lhs,block const& rhs) noexcept;
  friend bool operator==(block const& lhs,block const& rhs) noexcept
  {
    return lhs.data == rhs.data && lhs.length == rhs.length;
  }

  void reset() noexcept { *this = block{} }
  void swap(block& other) { /* ... */ }

  void*           data    = nullptr;  // Pointer to the allocated block of memory
  std::ptrdiff_t  length  = 0;        // Size in bytes of the allocated block of memory
};
```

Allocator design in Kiwaku

A Simple `malloc` allocator

```cpp
struct heap_allocator
{
  [[nodiscard]] block allocate(std::ptrdiff_t n) noexcept
  {
    return (n≠0) ? block{ malloc(n), n } : block{ nullptr, n };
  }

  void deallocate(block & b) noexcept { if(b.data) free(b.data); }

  void swap(heap_allocator&) {}
};
```

- No `virtual` interface
- No complex CRTP-like definition

Allocator design in Kiwaku

The `any_allocator`

- Uses Parent-style polymorphic object designs
- Distinct from `std::pmr::polymorphic_allocator` (no `memory_resource`)
- Provides an associated concept `kwk::concepts::allocator`

The Allocator Trifecta

- A virtual API object
- A template adapter implementing said API
- A `SemiRegularType` wrapper

Allocator design in Kiwaku

The `allocator` Concept

- We require some `allocate` and `deallocate` functions
- We expand upon `std::semiregular` and `std::swappable`

```cpp
template<typename A>
concept allocator =   std::semiregular<A>
                  &&  std::swappable<A>
                  &&  requires(A a, block& b, std::ptrdiff_t n)
{
  { a.allocate(n)   } -> std::same_as<block>;
  { a.deallocate(b) };
};
```

Allocator design in Kiwaku

The `any_allocator` - Basic Virtual API

- Only piece of polymorphism in the design
- Internal type to `kwk::any_allocator`

```cpp
struct api_t
{
  virtual ~api_t() {} // Obviously

  virtual block allocate(std::size_t) = 0;   // Actual allocator interface
  virtual void  deallocate(block&)    = 0;   // Actual allocator interface

  virtual std::unique_ptr<api_t>  clone() const = 0; // Helper for polymorphic copy
};
```

Allocator design in Kiwaku

The `any_allocator` - Template Adapter

- Final class implementing `api_t`
- Use `concepts::allocator` to prevent errors

```
template<concepts::allocator T> struct model_t final : api_t
{
  model_t() = default;
  model_t(const T&  t)  : object(t)             {}
  model_t(T&&       t)  : object(std::move(t))  {}

  block allocate(std::size_t n)        override { return object.allocate(n);                }
  void  deallocate(block& b)           override { object.deallocate(b);                      }
  std::unique_ptr<api_t> clone() const override { return std::make_unique<model_t>(object); }

  private:
  T object;
};
```

Allocator design in Kiwaku

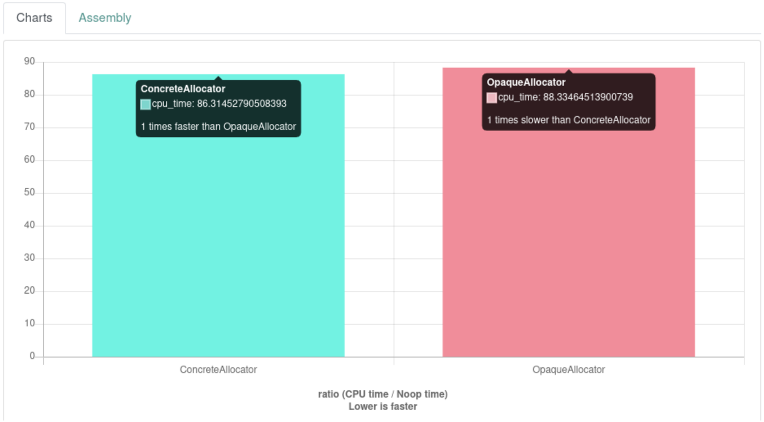
The `any_allocator` - `SemiRegularType` wrapper

```cpp
class any_allocator
{
  struct api_t  { /* ... */ };
  template<concepts::allocator T> struct model_t final : api_t { /* ... */ };

  std::unique_ptr<api_t> data;

  public:
  any_allocator(any_allocator const& a) : data( a.data->clone() ) {}

  // ... All other obvious special members

  template<typename T> any_allocator(T&& t) : data(make_model(std::forward<T>(t))) {}

  void swap(any_allocator& other) noexcept    { data.swap(other.data);    }
  [[nodiscard]] block allocate(std::size_t n) { return data->allocate(n); }
  void  deallocate(block& b)                  { data->deallocate(b);      }
};
```

Kiwaku Allocator - Benchmarks

Rough QuickBench

- Succession of allocate/deallocate of 16 Mb
- Scenario favorable to de-virtualization



ConcreteAllocator
🟦 cpu_time: 86.31452790508393
1 times faster than OpaqueAllocator

OpaqueAllocator
🟥 cpu_time: 88.33464513900739
1 times slower than ConcreteAllocator

ratio (CPU time / Noop time)
Lower is faster

Kiwaku Allocator - Benchmarks

More specific nanobench

- Multiple allocation of 16 Mb
- Single final deallocation
- Scenario unfavorable to de-virtualization

relative	ns/op	op/s	err%	Scenario
100.0%	4,627.60	216,094.74	8.1%	`Concrete Allocation`
101.6%	4,553.55	219,609.10	3.3%	`Opaque Allocation`

Kiwaku Allocator - Benchmarks

More specific nanobench

- Multiple allocation of 256 b
- Single final deallocation
- Scenario unfavorable to de-virtualization

relative	ns/op	op/s	err%	Scenario
100.0%	182.05	5,493,007.13	4.9%	`Concrete Allocation`
97.1%	187.53	5,332,397.82	4.0%	`Opaque Allocation`

Opaque Types - Conclusion

Simplify API by using Opaque Types

- Allocator are no longer parts of the template type of tables
- Less rigid template API
- Good candidate to be pre-compiled (consider using LTO?)

What do we learn

- Building interface as a set of concrete type + Parent's polymorphic type is a win
- Easy to maintain and to extend for users
- Do your homework and benchmark!

Tips #2 - Keyword Parameters

Keyword parameters - End goals

Kiwaku container constructors

```cpp
using namespace kwk::literals;

// Dynamic array using the default allocator
kwk::array<float,kwk::_2D> a1(kwk::of_shape(200,200))

// Dynamic array using the some other allocator modeling concepts::allocator
kwk::array<float,kwk::_2D> a2 ( "allocator"_kw  = some_allocator{}
                              , "shape"_kw      = kwk::of_shape(20,20)
                              );

// Allocator and data are copied to a1
a1 = a2;
```

Keyword Parameters

Definition

- Languages may provide a syntax to pass arguments to function based on their names
- Such parameters are called Keyword Parameters
- Ex: Python, C#

Challenges in C++

- Should they participate in mangling ?
- Which names of a parameters count ?
- See `N4172`

Keyword Parameters

Our Use Case

- Passing parameters to array or view constructors to simplify API
- Passing `constexpr` parameters to array or view constructors type interface
- Keyword can be predefined
- A function should be able to restrict which keyword it accepts

Our Solution

- RABERU: a library solution for keyword parameters
- Define keywords locally as `constexpr` instance of unique types
- Retrieve data from a keywords using a lambda as container
- Concepts can restrict the keyword to pass to a function

Keywords parameters as lightweight EDSL

Defining a keyword

- `rbr::keyword` acts as a keyword builder
- Keyword can be predefined
- The UDL syntax allow for local, on the spot keyword access

```cpp
namespace kwk::keyword
{
  // The rbr::keyword inline variable generate a new keyword_type
  inline constexpr auto shape = rbr::keyword<struct shape_option>;

  // The _kw UDL generate a keyword from the list of character of the string
  inline constexpr auto allocator = "allocator"_kw;

  // Equivalent without UDL
  inline constexpr auto allocator = rbr::keyword<id_<'a','l','l','o','c','a','t','o','r'>>;
}
```

Keywords parameters as lightweight EDSL

Binding a value to a keyword

- `keyword` has a generic assignment operator
- This operator returns a `linked_value` constructed from the keyword
- The `linked_value` is initialized with a lambda capturing the value of the parameters
- This lambda accept the keyword as a parameter and return the value

```
some_function(shape = extent[4][6]);
```

Retrieving a value from a keyword

- All keyword/value pairs are gathered in a `overload` like structure
- Every `operator()` of each pair is put back into the interface
- Fetching a value is simply done by calling this overload with the required keyword

Keywords parameters as lightweight EDSL

Binding a value to a keyword

```cpp
template<typename T> template<typename V>
constexpr auto keyword_type<T>::operator=(V &&v) const noexcept
{
  using type = keyword_type<T>;
  if constexpr( std::is_lvalue_reference_v<V> )
  {
    return linked_value(*this, [&v](type const &) -> decltype(auto) { return v; });
  }
  else
  {
    return linked_value ( *this
                        , [w = std::move(v)](type const &) -> V const & { return w; }
                        );
  }
}
```

Keywords parameters as lightweight EDSL

Retrieving a value from a keyword

```cpp
// Notify of an unsupported keyword
struct unknown_key { template<typename... T> unknown_key(T &&...) {} };

// Aggregate lambdas and give them a operator(Key)-like interface
template<typename... Ts> struct aggregator : Ts...
{
  constexpr aggregator(Ts &&...t) noexcept : Ts(RBR_FWD(t))... {}
  using Ts::operator()...;

  template<typename K> constexpr auto operator()(keyword_type<K> const &) const noexcept
  {
    // If not found before, return the unknown_key value
    return unknown_key {};
  }
};
```

Bringing everything together

The `settings` helper

- `settings` takes care of type deduction from a pack of keyword parameters
- It provides function to detect a keyword in a list of keyword parameters
- It provides function to validate a list of keyword parameters
- It supports optional default value if a keyword is not found

`keyword_parameter` and `match`

- Allow for proper constraint of function with keyword parameters
- Enable non-trivial requires clause based on the presence of a given keyword

Bringing everything together

The `settings` helper

```cpp
template<typename P0, typename P1>
auto replicate( P0 p0, P1 p1 )
{
  using namespace rbr::literals;
  auto const params = rbr::settings(p0,p1);

  return std::string( params["replication"_kw], params["letter"_kw] );
}

std::cout << replicate( "replication"_kw = 9, "letter"_kw = 'Z' ) << "\n";
```

Ouput:

```
ZZZZZZZZZ
```

Bringing everything together

The `settings` helper

```cpp
template<typename... Params>
auto replicate( Params... ps )
{
  using namespace rbr::literals;
  auto const params = rbr::settings(ps...);

  return std::string( params["replication"_kw | 5  ]
                    , params["letter"_kw      | '*']
                    );
}

std::cout << replicate( "letter"_kw = 'Z' ) << "\n";
```

Ouput:

```
ZZZZZ
```

Bringing everything together

The `keyword_parameter` concept

```cpp
template<rbr::keyword_parameter... Params>
auto replicate( Params... ps )
{
  using namespace rbr::literals;
  auto const params = rbr::settings(ps);

  return std::string( params["replication"_kw | 5   ]
                    , params["letter"_kw      | '*']
                    );
}

std::cout << replicate( "replication"_kw = 6 ) << "\n";
std::cout << replicate( 3.64 ) << "\n"; // won't compile
```

Ouput:

```
******
```

Bringing everything together

The `match` helper

```cpp
template<rbr::keyword_parameter... Params>
requires( rbr::match<Params...>::with("replication"_kw | "letter"_kw) )
auto replicate( Params... ps )
{
  using namespace rbr::literals;
  auto const params = rbr::settings(ps);

  return std::string( params["replication"_kw | 5  ]
                    , params["letter"_kw      | '*']
                    );
}

std::cout << replicate( "replication"_kw = 6 ) << "\n";
std::cout << replicate( "repilcation"_kw = 7 ) << "\n"; // won't compile
```

Ouput:

```
******
```

Keyword Parameters - Conclusion

Flexible API with Keyword Parameters

- Isolate common use cases from power users concerns
- Future proof and resistant to “oops I need to break the API” scenarios
- Compile cost low due to `if constexpr` and Concepts

What do we learn

- Keyword parameters features set can be tailored to fit C++
- Keyword parameters can be implemented in C++ now as a library
- Try Raberu at `https://github.com/jfalcou/ofw`

Tips #3 - Generic NTTP

Definition

```
template <class T, int N> array
```

Non-Type Template Parameter (NTTP) Before C++20

- An integral type
- An enumeration type
- A pointer type
- A pointer to member type
- `std::nullptr_t`
- A lvalue reference type

Definition

```
template <class T, int N> array
```

Non-Type Template Parameter (NTTP) Since C++20

- An integral type
- An enumeration type
- A pointer type
- A pointer to member type
- `std::nullptr_t`
- A lvalue reference type

- **A floating-point type**
- **A literal class type (with some restrictions)**

Opening a new era of template metaprogramming

```
template <auto Value> class_type
```

Generic NTTPs + Expression Template = EDSL mini-compilers

- EDSL = Embedded Domain-Specific Language
- Capture arbitrary constexpr expression as NTTP
- Process them to generate a proper implementation

```
template <auto Expression> edsl_compiler
```

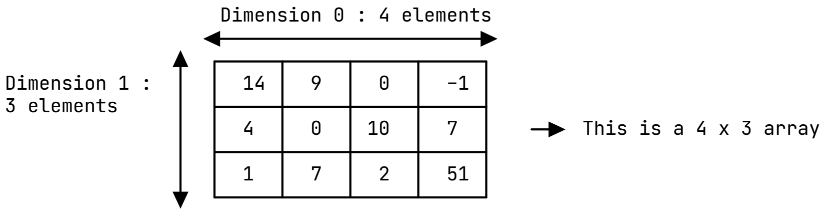
Challenge

- Defining array shapes
- Supporting both runtime, compile-time and hybrid shapes
- Plot Twist : With A Single Type!

Defining an array shape

Context

- Arrays gather data in a n-dimensional grid
- The number of effective dimensions is supposed **known at compile time**
- The number of elements along each dimension may vary
- The number of elements along a given axis may be known at compile time
- The initial ordering of those sizes is **domain specific** and **arbitrary**



Removing code duplication with NTTPs

Handling static + runtime sizes of arrays

- Option 1 : One type for shape, one for compile-time shape
- Option 2 : Verbose template hell

```
using my_span = std::mdspan<double, extents<3, 9, 7>>;
using my_other_span = std::mdspan<double, extents<3, std::dynamic_extent, 7>>;
```

Semantic-rich constexpr objects as NTTPs

- If we want a compile-time shape, we make a `constexpr` shape
- High-Performance: `constexpr` AST manipulation and optimization
- Generic: unified interfaces (static/dynamic sizes)
- Expressive: terse + precise

Array shapes as NTTP

Main idea

- Design a `extent` type only caring about runtime size storage
- Make it usable as an NTTP
- Provides helpers to smooth definition of array

What we want to achieve

```
array<float, _3D>                x;                        // Uninitialized 3D array with dynamic size
array<float, _2D>                y( 4, 6 );                // 2D array with dynamic size of 4x6
view<float, extent[4][3][1][2]> z(y.data());               // 4D view of y with static size of 4x3x1x2

array<float, extent[4]()[3]> a;                            // Uninitialized 3D array with size of 4x?x3
array<float, extent[4]()[3]> b( _[1] = 6 );                // 3D array with size of 4x6x3

constexpr auto               s = extent()();               // 2D dynamic extent
array<float, s[10]>          w;                            // Uninitialized 3D array with size of ?x?x10
```

Array shapes as NTTP

Benefits

- Unique type for static and dynamic extents
- `sizeof(array)` and `sizeof(view)` are minimal
- Safer and more expressive interface

What we want to achieve

```cpp
array<float, _3D>                x;                  // Uninitialized 3D array with dynamic size
array<float, _2D>                y( 4, 6 );          // 2D array with dynamic size of 4x6
view<float, extent[4][3][1][3]> z(y.data());         // 4D view of y with static size of 4x3x1x2

array<float, extent[4]()[3]> a;                      // Uninitialized 3D array with size of 4x?x3
array<float, extent[4]()[3]> b( _[1] = 6 );          // 3D array with size of 4x6x3

constexpr auto             s = extent()();           // 2D dynamic extent
array<float, s[10]>        w;                        // Uninitialized 3D array with size of ?x?x10
```

Shaper - Shaped definition Expression-template

Challenge: Building a Shape with memory of its construction

- Operator overloading for `()` and `[]`
- Incrementally build the data storage of size
- Provides helpers to access said data

```cpp
template <typename... Ds> struct shaper
{
  // ...
  template <typename... X> constexpr auto append(X... x);

  template <typename... Args>
  constexpr shaper(shaper<Args...> other, index_t i) : data_(other.append(i)) {}

  constexpr shaper<Ds...,dynamic_size> operator()()           const { return {*this, -1}; }
  constexpr shaper<Ds...,static_size>  operator[](index_t i)  const { return {*this,  i}; }

  std::array<index_t, sizeof...(Ds)> data_;
};
```

Extent - A Seed to grow shape trees

Defining extent definition helpers

```cpp
inline constexpr detail::shaper extent = {};

// Dynamic pre-rendered dimension shaper
inline constexpr auto _0D = extent;
inline constexpr auto _1D = extent();
inline constexpr auto _2D = extent()();
inline constexpr auto _3D = extent()()();
inline constexpr auto _4D = extent()()()();

// Dynamic nD short-cut
template<std::size_t N>
inline constexpr auto _nD = []<std::size_t... I>(std::index_sequence<I...> const&)
{
  return detail::shaper<decltype(detail::dynamic_size(I))...>{};
}(std::make_index_sequence<N>{});

// Some static shortcuts
inline constexpr auto _3x3        = extent[3][3];
inline constexpr auto _4x4        = extent[4][4];
inline constexpr auto rubiks_cube  = extent[3][3][6];
```

Shape - Runtime storage for extent

Challenges

```cpp
template<auto Shaper> struct shape
{
  // Provide a compact storage for only runtime dimensions
  // ???

  // Proper lifecycle and construction API
  shape(std::convertible_to<std::ptrdiff_t auto... );
  shape(std::same_as<axis> auto... );

  // Proper access to the data in all cases (CT,RT,hybrid)
  template<std::size_t I> constexpr std::ptrdiff_t get() const;
};
```

Shape - Runtime storage for extent

Challenge: Optimal storage and retrieval

- Exploit the structure of the `extent` object
- Build a compile-time bitmap of index where size is known at compile-time
- Store only the limited amount of size informations

```cpp
template<auto Shaper> struct shape
{
  using size_map = decltype(Shaper.size_map());
  static constexpr std::ptrdiff_t static_size = Shaper.size();
  static constexpr std::ptrdiff_t storage_size = static_size - size_map::size;

  using storage_type = std::array<std::ptrdiff_t,storage_size>;

  static constexpr bool is_dynamic = storage_size >= 1;
  static constexpr bool is_fully_dynamic = storage_size == static_size;
  static constexpr bool is_fully_static = storage_size == 0;

  storage_type data_;
};
```

Compile-time bitmap

Prototype and usage by `shape`

```cpp
template<typename... Ds> struct index_list
{
  // How many static dimensions ?
  static constexpr std::size_t size = (std::same_as<Ds,static_size> + ... );

  // is N a dimension we know at compile-time ?
  static constexpr bool contains(std::size_t N) noexcept

  // Find the runtime index of the Nth dimension runtime size
  template<std::size_t Size> static constexpr std::size_t locate(std::size_t N) noexcept;
};

template<auto Shaper>
template<std::size_t I> constexpr auto shaper<Shaper>::get() const noexcept
{
  if constexpr(size_map::contains(I))
    return std::integral_constant<std::ptrdiff_t,Shaper.at(I)>{};
  else
    return storage_[size_map::template locate<static_size>(I)];
}
```

Exploiting NTTPs in Generative Programming Context

View builder : Deducing shape and stride from settings

```cpp
template <typename Type, auto... Settings>
struct view : view_builder<Type,Settings...>::span
            , view_builder<Type,Settings...>::access
{ /* .... */ };

template <typename Type, auto... Settings>
struct view_builder
{
  static constexpr auto opt_    = rbr::settings(Settings...);

  static constexpr auto shape_  = kwk::shape< opt_[ "shape"_kw | _2D ] >{};
  static constexpr auto stride_ = opt_[ "stride"_kw | shape_.as_stride() ];

  static constexpr bool is_dynamic      = shape_.is_dynamic;
  static constexpr bool is_fully_static = shape_.is_fully_static;

  using span        = detail::view_span<Type*>;
  using access      = detail::view_access<shape_, stride_>;
};
```

Optimizing storage

A sample `view_access` specialization

```cpp
// Everything is static, don't store anything
// Expected sizeof of the view : sizeof(void*)
template<auto Shape, auto Stride>
requires( Shape.is_fully_static )
struct  view_access<Shape, Stride>
{
  using shape_type  = std::remove_cvref_t<decltype(Shape)>;
  using stride_type = std::remove_cvref_t<decltype(Stride)>;

  constexpr std::ptrdiff_t  size()    const noexcept  { return Shape.numel(); }
  constexpr auto            shape()   const noexcept  { return Shape;         }
  constexpr auto            stride()  const noexcept  { return Stride;        }

  template<typename... Int>
  constexpr std::ptrdiff_t index(Int... is) const noexcept { return Stride.index(is...); }

  void swap( view_access& other ) noexcept {}
};
```

Optimizing storage

A sample `view_access` specialization

```cpp
// Optimization : runtime 1D shape + unit stride
// Expected sizeof of the view : sizeof(void*) + sizeof(std::ptrdiff_t)
template<auto Shape, auto Stride>
requires( !Shape.is_fully_static && Shape.static_size == 1 && Stride.is_unit )
struct  view_access<Shape, Stride>
{
  using shape_type  = std::remove_cvref_t<decltype(Shape)>;
  using stride_type = std::remove_cvref_t<decltype(Stride)>;

  constexpr view_access( shape_type const& shp ) : shape_(shp) {}

  constexpr std::ptrdiff_t  size()                      const noexcept  { return get<0>(shape_);  }
  constexpr auto            shape()                     const noexcept  { return shape_;          }
  constexpr stride_type     stride()                    const noexcept  { return {};              }
  constexpr auto            index(std::ptrdiff_t is)    const noexcept  { return is;              }

  void swap( view_access& other ) noexcept { shape_.swap( other.shape_ ); }

  private:
  shape_type shape_;
};
```

Optimizing storage

Impact on code generation

```cpp
#include <kiwaku/container/view.hpp>

using namespace kwk;

void loop( view<float, extent[16][16]> v )
{
  for(std::ptrdiff_t i=0;i<v.size();++i)
   v(i) *= v(i) + 3;
}

void loop( float* v )
{
  for(std::ptrdiff_t i=0;i<16*16;++i)
   v[i] *= v[i] + 3;
}
```

Optimizing storage

Impact on code generation

- Raw C code is auto-vectorized
- Direct access to the data
- No excess bloat

```
loop(float*):
        movaps  xmm1, XMMWORD PTR .LC0[rip]
        lea     rax, [rdi+1024]
.L6:
        movups  xmm0, XMMWORD PTR [rdi]
        movups  xmm2, XMMWORD PTR [rdi]
        add     rdi, 16
        addps   xmm0, xmm1
        mulps   xmm0, xmm2
        movups  XMMWORD PTR [rdi-16], xmm0
        cmp     rax, rdi
        jne     .L6
        ret
```

Optimizing storage

Impact on code generation

- Kiwaku code is also auto-vectorized
- No excess bloat
- Size information is carried in the mangling

```
loop(view<float, shaper<static_size, static_size>{std::array<long, 2ul>{long [2]{16l, 16l}}}>):
        movaps  xmm1, XMMWORD PTR .LC0[rip]
        lea     rax, [rdi+1024]
.L2:
        movups  xmm0, XMMWORD PTR [rdi]
        movups  xmm2, XMMWORD PTR [rdi]
        add     rdi, 16
        addps   xmm0, xmm1
        mulps   xmm0, xmm2
        movups  XMMWORD PTR [rdi-16], xmm0
        cmp     rax, rdi
        jne     .L2
        ret
```

Generic NTTPs - Conclusion

Open new possibilities in terms of design

- Better APIs
- More generic interfaces
- Use of domain-specific information for high-levels of optimization

Multidimensional array shapes

- Non-incremental approach
- Unified static/dynamic array abstraction
- Terse, rich, and natural syntax
- High-levels of optimization for numerical arrays

Consider generic NTTPs + Expression Template as EDSL compilers

Conclusion

Summing up

The Times, They are-a Changing

- Like for C++11/14, C++17/20 is a game changer
- We now have tools to have better structured template code
- We can't go there by just incrementally changing existing code and practices

Lessons Learnt

- Breaking the old patterns was fruitful
- API design improved by using user-centric mindset
- No noticable drop in performances

What's next

Funky C++ Libraries and where to find them

- Raberu : The Keyword Parameters library
  - Part of `https://github.com/jfalcou/ofw`
  - Released and kind stable

- Kiwaku: Containers Done Right :
  - `https://github.com/jfalcou/kiwaku`
  - Still in pre-beta
  - Documentation pending

Looking forward

- Kind genericity
- Circle like reflection ?

Thanks for your attention !