Below is a **technical, line-by-line–aware review** of your multidimensional range-checked array design. Overall: **this is a clean, well-structured, and conceptually sound implementation** with a clear educational and safety-oriented intent. I'll focus on **correctness, API design, modern C++ practices, performance, and extensibility**, without drifting into over-engineering.

---

1. High-Level Assessment

Strengths

- ✔ Clear dimensional semantics (`dim` propagation is correct and intuitive)
- ✔ Compile-time sizes (no heap, no runtime allocation)
- ✔ Strong bounds checking with informative exceptions
- ✔ Zero-overhead abstraction when bounds checks are optimized away
- ✔ Clean recursive template composition (A1D ⟶ A2D ⟶ A3D ⟶ A4D)

Design intent clarity This is **not** a flat contiguous tensor abstraction; it is a **hierarchical array-of-arrays** model that preserves natural indexing and dimension identity. That intent is consistently respected throughout.

---

2. `RangeException` Review

What's good

```
struct RangeException : public std::out_of_range {
```

- Correct base class
- Rich diagnostic context (`dim`, `low`, `high`, `index`)
- Message construction is clean and readable

Minor improvements

2.1 Avoid string allocation on hot paths

Currently:

```
std::out_of_range(build_message(...))
```

This allocates a `std::string`. That's fine for debugging and correctness, but for performance-critical code you might consider:

- Lazy message generation
- Or storing raw values only and formatting on demand

But **this is completely acceptable** given the explicit safety orientation.

2.2 `constexpr`?

You cannot make this fully `constexpr` due to `std::string`, so current design is appropriate.

---

3. `A1D` — Core Abstraction

This is the **most important part of the design**, and it's done well.

3.1 Template parameters

```
template <typename Type, std::size_t Size, int D = 1>
```

- `Size` as non-type template parameter ⟶ compile-time bounds ✔
- `D` as a **dimension label**, not a rank ⟶ subtle but correct ✔

This is an excellent design choice:

> `D` represents **which axis the user indexed into**, not total dimensionality.

3.2 Index bounds logic

```
reference operator[](std::size_t i) {
    if (i >= Size)
        throw RangeException(D, 0, Size - 1, i);
    return element[i];
}
```

✔ Correct ✔ Exception contains the correct dimension index ✔ No undefined behavior

Edge case: `Size == 0`

- You guard `last_index()` but not the exception message:

  ```
  Size - 1  // underflows when Size == 0
  ```

Recommendation (important)

Add a static constraint:

```
static_assert(Size > 0, "Array dimension size must be > 0");
```

Or handle zero-sized arrays explicitly (but static_assert is cleaner).

3.3 `begin()` / `end()` semantics

```
constexpr std::size_t begin() const noexcept { return first_index(); }
constexpr std::size_t end() const noexcept { return last_index(); }
```

⚠️ **Naming issue**

These **do not follow STL iterator semantics**:

- `end()` should be **one-past-the-last**
- These return index bounds, not iterators

Recommendation

Either:

- Rename to `first_index()` / `last_index()` only
- Or redefine:

  ```
  constexpr std::size_t end() const noexcept { return Size; }
  ```

As written, `for (i = begin(); i != end(); ++i)` will **skip the last element**.

3.4 Storage

```
Type element[Size];
```

✔️ Correct ✔️ No overhead ✔️ Cache-friendly per dimension

---

4. Multidimensional Composition (`A2D`, `A3D`, `A4D`)

This is the most elegant part of your design.

4.1 Dimension mapping correctness

Example:

```
class A2D : public A1D<A1D<Type, size_2, 2>, size_1, 1>
```

This ensures:

Expression	Checked dimension
`arr[i]`	dim = 1
`arr[i][j]`	dim = 2

✔ **Perfect mapping** ✔ No off-by-one confusion ✔ Clean mental model

Same correctness propagates through `A3D` and `A4D`.

4.2 API consistency

- Exposed `SizeN` constants are useful and consistent
- `Dim` correctly expresses total dimensionality

Minor note: `Dim` is redundant since it's never used by the base class, but it's fine for introspection and documentation.

---

5. Performance Considerations

Runtime

- Bounds checks are minimal and branch-predictable
- Fully inlineable
- No virtual dispatch
- No heap allocation

Memory layout

- This is **array-of-arrays**, not a flat contiguous tensor
- Cache locality is good within the innermost dimension
- Slight pointer chasing across dimensions (acceptable trade-off)

If your intent were BLAS-style numeric kernels, a flat layout would be better — but that is **not** what this design targets.

---

6. Modern C++ Improvements (Optional)

These are **optional enhancements**, not corrections.

6.1 `constexpr` indexing (C++20)

You can make `operator[]` `constexpr`:

```
constexpr reference operator[](std::size_t i)
```

This allows compile-time evaluation for constant indices.

6.2 `[[nodiscard]]` on size helpers

```
[[nodiscard]] static constexpr std::size_t size() noexcept;
```

Prevents accidental ignoring of compile-time size info.

6.3 Concepts / Constraints (C++20)

If you generalize later:

```
static_assert(std::is_default_constructible_v<Type>);
```

Only if you want to constrain storage semantics.

---

7. Design Limitations (Explicit but Acceptable)

These are **not flaws**, just known boundaries:

- ❌ No slicing or views
- ❌ No iterator support
- ❌ No runtime resizing
- ❌ Fixed rank per type

Given your intent, all of these are **appropriate omissions**.

---

8. Overall Verdict

Quality: **Very High**

This code is:

- Conceptually clean
- Correctly implemented
- Exception-safe
- Pedagogically excellent
- Practically usable

Recommended minimal fixes

1. `static_assert(Size > 0)`
2. Fix or rename `begin()` / `end()`

Everything else is **solid and defensible as-is**.

If you want, I can:

- Generalize this to **N-dimensions via variadic templates**
- Convert it to a **flat contiguous tensor with the same diagnostics**
- Add **constexpr compile-time bounds checking**
- Or formally document this as a **library-grade component**

Just tell me which direction you want to go.