Chapter 1 — Type Erasure

Introduction

I've used type erasure in C++ for years without giving it much thought. For example, std::function is a well-known polymorphic, type erased template class that holds a callable object—any function, lambda, or functor. The concrete type of the callable is "erased"; it's hidden from the user (I know this sounds weird, but it will make sense later). What remains is a uniform interface to invoke it. The classes std::any and std::shared_ptr are other good examples of type erasure.

In their book *C++ Template Meta-programming: Concepts, Tools, and Techniques from Boost and Beyond*, Dave Abrahams and Aleksey Gurtovoy define type erasure as "the process of turning a wide variety of types with a common interface into one type with that same interface." This is an apt description.

The excellent book *Software Design in C++* by Klaus Iglberger provided me with a solid foundation in the mechanics of type erasure. Chapter 7 lays the groundwork, and Chapter 8 dives into type erasure in detail, including several performance optimizations. I leave it to the reader to obtain the book and do some studying; there is a lot of valuable learning there about design patterns and modern C++ in addition to type erasure.

In practice, type erasure is implemented as a combination of several different design patterns; according to Software Design in C++: External Polymorphism, Bridge, and optionally *Prototype*. I will not go into the details of these patterns here, but will point them out in the code that is presented in this chapter.

Here is a quick refresher on the three patterns:

- External Polymorphism – Moves the polymorphic interface outside the concrete types. Instead of inheritance, a separate "operation table" knows how to work with any stored type.
- Bridge – Splits the abstraction (the uniform interface) from the implementation (the underlying functionality), allowing them to vary

independently. The abstraction maintains a reference to the implementation and delegates work to it.

- Prototype (Optional) – Lets you clone or copy objects without knowing their concrete type by providing a generic “duplicate me” operation. This pattern is optional because you may not want your objects to be copyable.

Why is type erasure important in our example system? Because it allows us to treat unrelated types uniformly without forcing them into an inheritance hierarchy. As long as the stored types implement the expected concept (in terms of available member functions), they can be used interchangeably.

The important point here is that we can use these types with value semantics - no pointer-to-base-class is needed. A type-erased class provides a consistent interface to all stored types. In that sense, a type-erased class represents a concept, not a concrete type or explicit interface.

In this chapter, we will build a minimal type-erased representation of a system measurement. This allows heterogeneous measurement data to flow cleanly through the system with value semantics, without forcing concrete types into a common inheritance hierarchy.

Code for this chapter is in Github at https://github.com/AutumnalSoftware/BuildingEmbeddedSystemsOnLinuxBook/tree/main/Chapter4-TypeErasure

When Not to use Type Erasure

Type erasure is a powerful tool, but not always appropriate to a particular purpose.

Don’t use type erasure when:

- A simple template will do
- If call sites are known at compile time and you do not need runtime polymorphism, templates are usually more clear, faster, and easier to reason about.
- There is a natural and stable inheritance hierarchy
- If your domain has an “is-a” relationship that is unlikely to change, classic runtime polymorphism may be the simpler solution.
- You need the simplest possible abstraction and can tolerate tight coupling
- Many type-erased classes rely on heap allocation and virtual inheritance internally. In non-embedded systems, this overhead is acceptable. As Klaus

Iglberger in his book *C++ Software Design* shows, type-erased designs can be optimized to eliminate virtual dispatch and manage memory more explicitly.

- However, if you are running in hot paths or tight inner loops and can accept tight coupling, a more direct and concrete design may be the most practical.

Non-Goals for This Chapter

This chapter focuses only on using type erasure to represent heterogeneous system measurement types uniformly, with value semantics.

It does not address serialization, transport, threading, or message queues. These concerns introduce additional design elements: buffers, ownership, ordering, and timing that would distract from the core topic. They will be addressed explicitly in later chapters, once the underlying data model is established.

Copying and the Prototype Pattern

Although *AnyMeasurement* stores its implementation behind a pointer, it still behaves like a value type. This is achieved through the `clone()` function on *IMeasurementModel*.

When an *AnyMeasurement* is copied, it does not copy the pointer. Instead, it asks the underlying IMeasurementModel object to clone itself. Each *Model*<*T*> knows how to make a deep copy of its stored measurement, because it knows the concrete type.

This is a direct application of the *Prototype* pattern. It allows the wrapper to support copy semantics without exposing the concrete type or relying on inheritance in the public API.

From the user's perspective, copying an *AnyMeasurement* looks and behaves like copying any other value type.

Where Polymorphism Actually Lives

It is worth emphasizing where runtime polymorphism exists in this design.

Polymorphism lives only inside *AnyMeasurement*, within the private *IMeasurementModel* and *Model*<*T*> types.

There is no polymorphism in the concrete measurement types that we want to store, nor in user code or the public interface.

This is the defining characteristic of external polymorphism. The polymorphic behavior is moved out of the concrete types and into a separate abstraction layer. The result is a clean separation of concerns. Concrete measurement types model data and validation logic. The type-erased wrapper models uniform handling and ownership.

Adapting Concrete Measurement Types

In this version of the design, concrete measurement types are adapted to the erased interface by wrapping them in a type-specific model class.

For each concrete measurement type T, the wrapper instantiates a corresponding *Model*<*T*> that implements the erased interface (*IMeasurementModel*) by delegating operations to the stored value. This adaptation does not require inheritance, traits, or helper templates. Concrete measurement types remain plain data structures with no knowledge that type erasure is being used.

The adaptation occurs at runtime through virtual dispatch. If a concrete type does not support the operations required by the model, such as being printable for debugging, the error surfaces naturally at compile time when the model is instantiated.

This approach keeps the design intentionally simple. More sophisticated compile-time adaptation techniques are possible and will be explored later, but they are deliberately avoided here to keep the mechanics of type erasure easy to understand.

Why This Design Scales Well

Adding a new measurement type does not require modifying *AnyMeasurement*. As long as the new type provides the required operations, it automatically satisfies the erased concept.

There are no base classes to inherit from. There are no virtual functions in the measurement types. There is no registration step. Concrete measurement types remain as plain data structures and the abstraction scales naturally as the system grows.

This design provides a concrete example of the *Open-Closed Principle (OCP)*. AnyMeasurement is closed for modification. Adding new measurement types requires no changes to its implementation. The design is open for extension because new measurement types are introduced by addition; no existing code is touched.

Using AnyMeasurement

With the type-erased wrapper in place, using heterogeneous measurement types becomes straightforward. Different concrete measurement structures can be stored in the same container and passed through the same interfaces without exposing their concrete types.

For example, a collection of system measurements can be represented as a simple sequence:

```
std::vector<AnyMeasurement> measurements;
```

Each element in the container may hold a different concrete measurement type, but consumers interact with them uniformly. Code that processes measurements no longer needs to know how they are represented internally. It depends only on the small, stable interface exposed by the type-erased wrapper.

This is the primary benefit of the design. It allows measurement handling logic to be written once, in terms of concepts, rather than repeatedly rewritten for each concrete type.

A Small Demonstration

```
int main()
{
    MeasurementHeader tempHeader;
    tempHeader.name = "outdoor_temp";
    tempHeader.units = "C";
    tempHeader.timestampUs = 1736539200000000ULL; // demo number
    tempHeader.type = MeasurementType::Temperature;
    tempHeader.quality = Quality::Good;

    MeasurementHeader pressureHeader;
    pressureHeader.name = "baro_pressure";
    pressureHeader.units = "kPa";
    pressureHeader.timestampUs = 1736539200100000ULL;
    pressureHeader.type = MeasurementType::Pressure;
    pressureHeader.quality = Quality::Good;

    MeasurementHeader posHeader;
    posHeader.name = "gps_fix";
    posHeader.units = "deg";
    posHeader.timestampUs = 1736539200123456ULL;
    posHeader.type = MeasurementType::Position;
    posHeader.quality = Quality::Suspect;
```

```
    AnyMeasurement temp =
        AnyMeasurement::make(tempHeader, Temperature{23.5});

    AnyMeasurement pressure =
        AnyMeasurement::make(pressureHeader, Pressure{101.3});

    AnyMeasurement pos =
        AnyMeasurement::make(posHeader, Position{42.93,
-77.54});

    std::cout << temp << "\n";
    std::cout << pressure << "\n";
    std::cout << pos << "\n";

    // Copy semantics
    AnyMeasurement copy = temp;
    std::cout << "copy:  " << copy << "\n";

    // Move semantics
    AnyMeasurement moved = std::move(pressure);
    std::cout << "moved: " << moved << "\n";

    // The moved-from object is valid but unspecified. We'll
just
    // show it.
    std::cout << "after move, pressure: " << pressure << "\n";

    return 0;
}
```

Looking Ahead

This chapter stopped short of addressing serialization, transport, threading, and message queues on purpose. Those concerns introduce additional dimensions of complexity that require focused treatment of their own.

The important point is that the design introduced here is ready for those next steps. A type-erased measurement with value semantics is an ideal unit for queuing, transport, logging, and testing. Later chapters will build on this foundation to move measurements between threads, serialize them for transport, and integrate them into the broader system pipeline.

By separating representation from transport, and data modeling from concurrency, the system evolves in layers rather than as a single tangled abstraction. This is a recurring theme throughout the book.

This MVP is correct and easy to understand, but it is not adequate for an embedded target with strict memory constraints. This implementation uses heap allocation and virtual dispatch internally. In later chapters, we will revisit this design with explicit memory strategies that avoid per-object heap allocation and allow tighter control over latency and footprint.

Chapter Summary

In this chapter, we introduced type erasure as a practical design technique for representing heterogeneous system measurement types uniformly. We built a minimal, value-semantics wrapper that avoids inheritance, keeps concrete measurement types simple, and scales cleanly as new measurement types are added.

The result is a small but powerful abstraction. It demonstrates external polymorphism in practice, enforces the Open–Closed Principle structurally, and provides a stable foundation for the more complex system behaviors that follow.