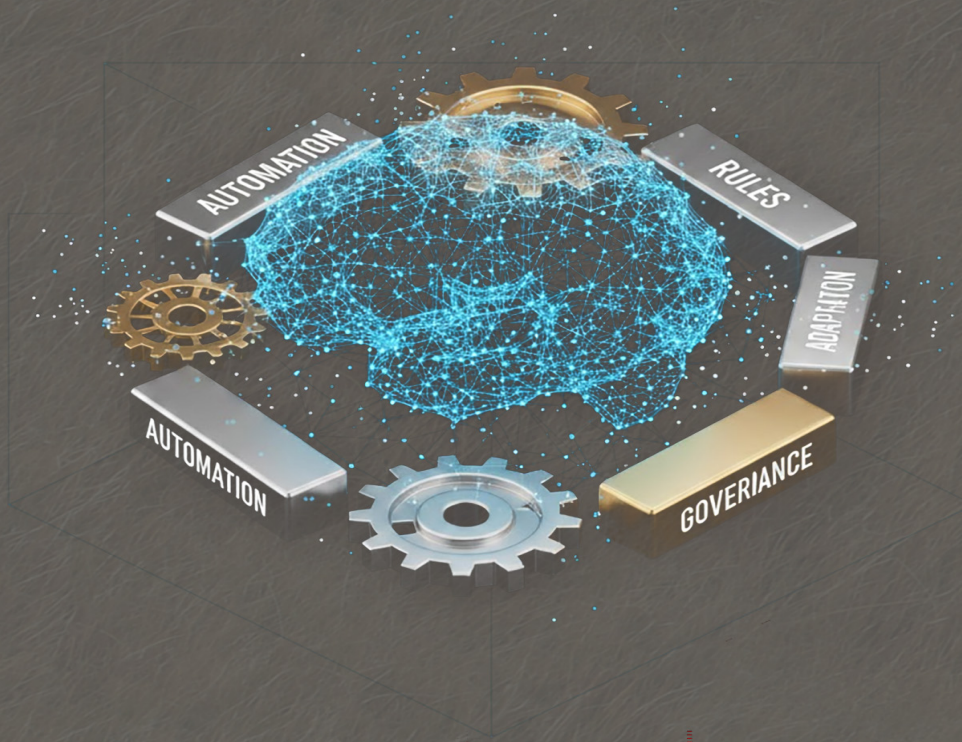
https://simplifycpp.org

Policy-Based Design

A Quick Understanding of Advanced Techniques



Prepared by Ayman alheraki

Policy-Based Design

A Quick Understanding of Advanced Techniques

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Contents **2**

Author's Introduction **9**

Preface **10**

1 The Design Problem Policy-Based Design Solves **12**

1.1 Why Design Matters More Than Syntax 12

1.2 The Classical Object-Oriented Trap 13

1.3 Rigid Hierarchies and Combinatorial Explosion 13

1.4 Runtime Cost as an Architectural Side Effect 14

1.5 Implicit Coupling and Hidden Dependencies 15

1.6 Why Composition Alone Is Not Enough 16

1.7 The Core Insight Behind Policy-Based Design 16

1.8 What This Booklet Will Build Upon 17

2 Compile-Time Polymorphism Revisited **18**

2.1 Polymorphism Beyond Virtual Functions 18

2.2 Static vs Dynamic Polymorphism 19

2.2.1 Dynamic Polymorphism 19

2.2.2 Compile-Time Polymorphism 20
2.3 Templates as Polymorphic Mechanisms 20
2.4 The Curiously Recurring Template Pattern (CRTP) 21
2.5 Why CRTP Alone Is Not Enough 22
2.6 The Zero-Overhead Principle 22
2.7 Type-Based Design Thinking 23
2.8 Preparing for Policies 24

3 What Is a Policy? 25
3.1 The Need for a Precise Definition 25
3.2 Policy vs Strategy 26
3.2.1 Strategy Pattern 26
3.2.2 Policy-Based Design 27
3.3 Policy as a Behavioral Contract 27
3.4 Orthogonality of Policies 28
3.5 Policies Are Not Configuration Objects 29
3.6 Policies as Building Blocks 30
3.7 What Policies Are Not 30
3.8 Looking Ahead 31

4 Type Policies and Traits 32
4.1 Why Types Can Carry Meaning 32
4.2 Type Policies as Compile-Time Decisions 32
4.3 Using Type Policies in Interfaces 33
4.4 Traits as Compile-Time Queries 34
4.5 Conditional Compilation with Traits 35
4.6 Type Policies vs Behavioral Policies 35
4.7 Tag Types and Intent Encoding 36

4.8 Scaling with Multiple Type Policies 37
4.9 Guidelines for Designing Type Policies 37
4.10 Preparing for Behavioral Policies 38

5 Behavioral Policies **39**
5.1 From Description to Action 39
5.2 Defining a Behavioral Policy 40
5.3 Injecting Behavior via Templates 40
5.4 Stateless vs Stateful Behavioral Policies 41
5.4.1 Stateless Policies 41
5.4.2 Stateful Policies 42
5.5 Passing Stateful Policies Safely 43
5.6 Behavioral Policies and Inlining 43
5.7 Combining Type and Behavioral Policies 44
5.8 Avoiding Policy Overreach 45
5.9 Testing Behavioral Policies 46
5.10 Looking Forward 46

6 Storage and Allocation Policies **47**
6.1 Why Memory Strategy Is an Architectural Decision 47
6.2 What Is a Storage Policy? 48
6.3 A Simple Dynamic Allocation Policy 48
6.4 A Static Storage Policy 49
6.5 Using a Storage Policy in a Class 50
6.6 Allocator-Based Storage Policies 51
6.7 Storage Policies and Exception Safety 52
6.8 Cache Locality and Embedded Storage 53
6.9 Avoiding Over-Engineering 53

6.10 Key Design Guidelines 54
6.11 Looking Ahead 54

7 Error Handling Policies **55**
7.1 Error Handling as a Design Choice 55
7.2 What Is an Error Handling Policy? 56
7.3 Exception-Based Error Policy 56
7.4 Error-Code-Based Policy 57
7.5 Fail-Fast Policy 58
7.6 Integrating Error Policies into Components 59
7.7 Error Policies and `noexcept` 59
7.8 Combining Error Policies with Other Policies 60
7.9 Testing Error Policies 60
7.10 Design Guidelines 61
7.11 Looking Ahead 61

8 Policy Composition **62**
8.1 Why Composition Is the Real Challenge 62
8.2 Independent Dimensions of Behavior 63
8.3 Layered Policy Application 63
8.4 Delegation Through Policy Interfaces 64
8.5 Ordering and Dependency Constraints 65
8.6 Avoiding Policy Interdependence 66
8.7 Managing Template Parameter Explosion 66
8.8 Policy Bundles 67
8.9 Defaults and Partial Customization 68
8.10 Compile-Time Validation of Policy Sets 68
8.11 Balancing Power and Clarity 69

8.12 Looking Ahead 69

9 Policy-Based Design vs Object-Oriented Design 71

9.1 Two Design Philosophies, Not Opponents 71
9.2 Strengths of Classical Object-Oriented Design 72
9.3 Limitations of Object-Oriented Design in C++ 72
9.4 Where Policy-Based Design Excels 73
9.5 Expressiveness vs Flexibility 74
9.6 Inheritance vs Composition by Types 74
9.7 Binary Interfaces and ABI Stability 75
9.8 Testing and Reasoning About Code 76
9.9 Hybrid Designs: Using Both Effectively 76
9.10 Design Smells and Warning Signs 77
9.11 Choosing the Right Tool 77
9.12 Looking Ahead 78

10 Policy-Based Design in Modern C++ (C++20/23) 79

10.1 Why Modern C++ Changes the Equation 79
10.2 Concepts as Explicit Policy Contracts 80
10.3 Constraining Policy Parameters 81
10.4 Replacing SFINAE with `requires` 81
10.5 `if constexpr` and Policy Logic 82
10.6 Stronger `constexpr` Capabilities 82
10.7 Improved Diagnostics and Maintainability 83
10.8 Reducing Template Noise 84
10.9 Constexpr-Friendly Policy Validation 84
10.10 Modern C++ Enables Safer Abstractions 85
10.11 Design Guidelines for Modern Policy-Based Code 85

10.12Looking Ahead 85

11 Testability and Maintainability **87**

11.1 Why Architecture Determines Testability 87

11.2 Explicit Dependencies by Construction 88

11.3 Testing Policies in Isolation 89

11.4 Mock Policies Instead of Mock Objects 89

11.5 Compile-Time Validation Reduces Test Burden 90

11.6 Deterministic Behavior Simplifies Testing 91

11.7 Improved Refactorability 91

11.8 Reducing the Need for Integration Tests 92

11.9 Long-Term Maintainability 92

11.10Avoiding Test Fragility 93

11.11Design Guidelines for Testable Policies 94

11.12Looking Ahead 94

12 A Real-World Case Study: From Inheritance to Policies **95**

12.1 Why a Case Study Matters 95

12.2 The Original Inheritance-Based Design 96

12.3 Problems Observed in Practice 97

12.4 Identifying Orthogonal Concerns 98

12.5 Defining the Policies 98

12.5.1 Logging Policies 98

12.5.2 Error Handling Policies 99

12.5.3 Storage Policy 99

12.6 Rebuilding the Processor with Policies 99

12.7 Instantiating Concrete Configurations 100

12.8 Testing Becomes Trivial 101

12.9 Performance and Clarity Gains 101
12.10Lessons Learned 102
12.11Looking Ahead 102

Appendices **103**
Appendix A: Common Anti-Patterns in Policy-Based Design 103
Appendix B: Policy Design Checklist 108

Conclusion **112**

References **114**

Author's Introduction

This booklet was written for C++ engineers who have moved beyond syntax and are beginning to confront the true challenges of large-scale software design.

At a certain level of experience, bugs are no longer the primary problem. Architectural rigidity, hidden coupling, and long-term maintainability become far more costly than individual implementation mistakes.

Policy-Based Design is often encountered at this stage. It appears in advanced libraries, performance-critical systems, and foundational infrastructure code. Yet it is frequently misunderstood, avoided, or misapplied.

Some view it as unnecessary template complexity. Others attempt to use it without understanding the design problems it solves, resulting in unreadable and fragile code.

This booklet was written to correct that gap.

Policy-Based Design is not about clever templates. It is about architectural clarity. It is about making decisions explicit, enforceable, and visible to the compiler. It is about designing systems that scale in both performance and understanding.

The techniques presented here are not theoretical. They are drawn from real-world C++ systems that must survive years of change, growth, and maintenance.

Ayman Alheraki

Preface

This booklet focuses on one central idea:

> Most behavioral variation in software systems is architectural, not dynamic—and should be expressed as such.

In traditional object-oriented design, variation is often encoded through inheritance and virtual functions. While effective in some domains, this approach becomes brittle when applied indiscriminately, especially in performance-sensitive or library-level code.
Policy-Based Design offers an alternative. It shifts variation from runtime to compile time, from implicit inheritance to explicit composition, and from mutable configuration to type-driven intent.
This booklet is not a general introduction to templates. It assumes the reader is already comfortable with modern C++ fundamentals. Its purpose is to demonstrate how templates, when used deliberately and with restraint, enable cleaner, safer, and more maintainable architectures.
What this booklet **is**:

- A focused architectural guide
- A practical explanation of policy-based techniques
- A modern interpretation aligned with C++20 and C++23

What this booklet **is not**:

- A catalog of template tricks
- A replacement for object-oriented design
- A framework or coding standard

Readers are encouraged to move slowly, reflect on each design trade-off, and apply these techniques selectively. Policy-Based Design is powerful, but only when guided by judgment and clarity.

Chapter 1

The Design Problem Policy-Based Design Solves

1.1 Why Design Matters More Than Syntax

C++ offers an enormous range of abstraction mechanisms. Templates, inheritance, virtual functions, inline functions, and compile-time evaluation give engineers expressive power unmatched by most languages.
However, this power introduces a paradox:

> The more expressive the language, the more damaging poor design becomes.

In small programs, architectural mistakes are tolerable. In large, long-lived systems, they become permanent liabilities.
Policy-Based Design exists not to showcase template cleverness, but to address structural design failures that emerge when traditional object-oriented techniques are applied beyond their limits.
This chapter focuses on the *problem space*, not the solution. Understanding why Policy-Based Design exists is essential before attempting to use it.

1.2 The Classical Object-Oriented Trap

Object-Oriented Programming encourages modeling behavior through inheritance and runtime polymorphism.
A typical design starts innocently:

- A base class defines a common interface
- Derived classes override virtual functions
- Behavior is selected at runtime

This approach works well for:

- User interfaces
- Plugin systems
- Runtime-extensible architectures

Problems arise when OOP is used as a *default solution* instead of a *contextual tool*.

1.3 Rigid Hierarchies and Combinatorial Explosion

Consider a system with multiple independent dimensions of behavior:

- Logging strategy
- Error handling strategy
- Memory allocation strategy
- Thread-safety model

Using inheritance, each variation often leads to a new derived class.
Very quickly, the number of required classes grows exponentially:

- Logged + Exception-based
- Logged + Error-code-based
- Silent + Exception-based
- Silent + Error-code-based
- Thread-safe + Logged + Exception-based
- ...

This phenomenon is known as *combinatorial explosion*.
Inheritance forces unrelated design decisions into a single hierarchy, making the system fragile and resistant to change.

1.4 Runtime Cost as an Architectural Side Effect

Virtual functions introduce:

- Indirection
- Loss of inlining
- Reduced optimization opportunities

While modern compilers mitigate some of these costs, the deeper issue is architectural:

> Runtime polymorphism forces decisions that could be static to remain dynamic.

When behavior is known at compile time but implemented through runtime dispatch, the design violates C++'s zero-overhead principle.
This is not merely a performance issue. It is a signal that the abstraction level is mismatched to the problem.

1.5 Implicit Coupling and Hidden Dependencies

Inheritance introduces tight coupling:

- Base classes dictate structure
- Derived classes inherit more than intended
- Changes ripple unpredictably

Developers often describe this as:

> "I changed one virtual function and broke unrelated code."

This happens because inheritance:

- Mixes interface with implementation
- Encourages shared mutable state
- Obscures ownership and responsibility

Policy-Based Design seeks to make such dependencies explicit and localized.

1.6 Why Composition Alone Is Not Enough

Modern C++ encourages composition over inheritance. While this is a step in the right direction, composition alone does not solve:

- Configuration explosion
- Boilerplate forwarding
- Loss of compile-time guarantees

Without policy abstraction, composition often degenerates into:

- Deep constructor parameter lists
- Verbose delegation code
- Runtime configuration objects

Policy-Based Design extends composition into the compile-time domain, where decisions are explicit, enforced, and optimized.

1.7 The Core Insight Behind Policy-Based Design

The key realization is simple but powerful:

> Most behavioral variation in systems is static, not dynamic.

If behavior:

- Does not change at runtime
- Is known at design time

- Can be expressed as a type

Then it should be:

- Selected at compile time
- Enforced by the type system
- Optimized away by the compiler

Policy-Based Design formalizes this idea.

1.8 What This Booklet Will Build Upon

The remainder of this booklet will show:

- How policies replace rigid inheritance
- How compile-time composition scales
- How modern C++ simplifies policy constraints
- How to write policy-driven code that remains readable

Before introducing templates, traits, or concepts, this chapter establishes a critical foundation:

> Policy-Based Design is not an advanced trick. It is a disciplined response to architectural failure.

The next chapter revisits compile-time polymorphism and explains why it is the natural foundation for policy-based systems.

Chapter 2

Compile-Time Polymorphism Revisited

2.1 Polymorphism Beyond Virtual Functions

When programmers hear the term *polymorphism*, they often associate it exclusively with inheritance and virtual functions. This association is understandable, as classical object-oriented education presents runtime polymorphism as the primary mechanism for behavioral variation.

C++ supports runtime polymorphism, but it was never limited to it. From its earliest days, C++ also supported a fundamentally different model: *compile-time polymorphism*.

Compile-time polymorphism does not rely on base classes, virtual tables, or runtime indirection. Instead, behavior is selected and resolved during compilation, allowing the compiler to reason about concrete types and generate optimal code.

Policy-Based Design is built entirely on this form of polymorphism.

2.2 Static vs Dynamic Polymorphism

To understand why policies favor compile-time mechanisms, we must clearly distinguish the two models.

2.2.1 Dynamic Polymorphism

Dynamic polymorphism is characterized by:

- A base class with virtual functions
- Derived classes overriding behavior
- Dispatch resolved at runtime

Its strengths include:

- Runtime extensibility
- Binary stability across modules
- Interface-based programming

Its costs include:

- Indirection through virtual tables
- Inhibited inlining
- Hidden coupling via inheritance

Dynamic polymorphism is a powerful tool, but it is not free, and it is not universally appropriate.

2.2.2 Compile-Time Polymorphism

Compile-time polymorphism is characterized by:

- Behavior selected by types
- No virtual functions
- Dispatch resolved during compilation

Its strengths include:

- Zero runtime overhead
- Aggressive optimization
- Stronger static guarantees

Its primary limitation is that behavior must be known at compile time. For many systems, this is not a limitation at all, but an advantage.

2.3 Templates as Polymorphic Mechanisms

Templates are often described as code-generation tools. This description is incomplete. Templates are better understood as *compile-time interfaces*. They allow code to operate on abstract behavior without prescribing a concrete inheritance hierarchy.
Consider a function template that operates on a type:

```
template <typename T>
void process(T& obj) {
    obj.execute();
}
```

Here, polymorphism exists without:

- A base class
- A virtual function
- A common ancestor

The only requirement is that `T` provides an `execute()` member. This requirement is structural rather than nominal.
Policy-Based Design relies heavily on this form of structural polymorphism.

2.4 The Curiously Recurring Template Pattern (CRTP)

One of the most important building blocks of compile-time polymorphism is the Curiously Recurring Template Pattern (CRTP).
CRTP allows a base class to refer to the derived class as a template parameter.

```
template <typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};
```

A derived class then provides the concrete behavior:

```
class Concrete : public Base<Concrete> {
public:
    void implementation() {
        // concrete behavior
    }
};
```

This pattern enables:

- Static dispatch
- Code reuse without virtual functions
- Compile-time binding of behavior

CRTP is not a policy system by itself, but it demonstrates how behavior can be deferred to derived types without runtime cost.

2.5 Why CRTP Alone Is Not Enough

While CRTP is powerful, it has limitations:

- It still imposes a base-class relationship
- It does not scale well across independent dimensions of behavior
- It entangles structure and behavior

Policy-Based Design generalizes the idea: instead of inheriting behavior, types *receive behavior as parameters*.
This subtle shift eliminates many of the structural constraints imposed by CRTP-based hierarchies.

2.6 The Zero-Overhead Principle

One of C++'s foundational design philosophies is the zero-overhead principle:

> What you do not use, you do not pay for. What you do use, you could not do better by hand.

Compile-time polymorphism aligns perfectly with this principle.
When behavior is:

- Selected by type
- Known at compile time
- Free of runtime indirection

The compiler can:

- Inline aggressively
- Eliminate dead code paths
- Optimize across abstraction boundaries

Policy-Based Design is not an optimization technique, but it naturally enables optimal code generation.

2.7 Type-Based Design Thinking

Compile-time polymorphism encourages a different way of thinking about design.
Instead of asking:

> What object should handle this behavior?

We ask:

> What type expresses this behavior?

This shift moves architectural decisions:

- From runtime to compile time

- From configuration files to type parameters
- From mutable state to immutable design

Policies are the formalization of this approach.

2.8 Preparing for Policies

By the end of this chapter, the key ideas should be clear:

- Polymorphism is not synonymous with inheritance
- Templates enable static behavioral variation
- Compile-time selection improves clarity and performance

The next chapter introduces the central abstraction of this booklet: the *policy* itself.
We will define what a policy is, what it is not, and how it differs from related design concepts such as strategies and traits.

Chapter 3

What Is a Policy?

3.1 The Need for a Precise Definition

The term *policy* is often used loosely in C++ discussions. Some use it to mean a configuration object. Others equate it with the Strategy pattern. In many codebases, it becomes a vague synonym for "something customizable."
For Policy-Based Design to be effective, the term must be precise.
In this booklet, a **policy** is defined as:

> A policy is a type that encapsulates a single, well-defined behavioral decision, intended to be selected and composed at compile time.

This definition deliberately emphasizes:

- **Type-based expression**
- **Single responsibility**
- **Compile-time selection**

Anything that violates these properties is not a policy in the architectural sense used throughout this book.

3.2 Policy vs Strategy

A common source of confusion is the relationship between policies and the Strategy design pattern.
Both aim to vary behavior. The difference lies in *when* and *how* that variation occurs.

3.2.1 Strategy Pattern

The Strategy pattern:

- Selects behavior at runtime
- Uses inheritance and virtual functions
- Stores behavior behind pointers or references

This makes it ideal for:

- User-driven configuration
- Plugin architectures
- Runtime adaptability

However, it also introduces:

- Runtime indirection
- Lifetime management complexity
- Reduced optimization opportunities

3.2.2 Policy-Based Design

Policies:

- Select behavior at compile time
- Use templates instead of inheritance
- Are composed through types

This makes them ideal for:

- Libraries
- Performance-critical systems
- Architecturally fixed behavior

A policy is not a replacement for a strategy. It is a solution to a different class of problems.

3.3 Policy as a Behavioral Contract

A policy defines a *behavioral contract*.
This contract is usually implicit and expressed through:

- Required member functions
- Required type aliases
- Expected semantics

For example, consider a logging policy:

```
struct NullLogger {
    static void log(const char*) noexcept {}
};
```

This policy expresses a clear intent:

- Logging is optional
- The call is safe
- The cost is zero

Another policy might provide actual logging:

```
#include <iostream>

struct ConsoleLogger {
    static void log(const char* msg) {
        std::cout << msg << '\n';
    }
};
```

Both types satisfy the same *policy contract*, despite having radically different behavior.

3.4 Orthogonality of Policies

One of the most important properties of good policy design is *orthogonality*.
An orthogonal policy:

- Solves one problem
- Does not depend on unrelated decisions
- Can be composed freely with others

For example:

- Logging policy

- Allocation policy
- Error handling policy

Each of these addresses a distinct concern. They should not know about each other. They should not share state. They should not impose ordering constraints.
Orthogonality is what prevents policy-based systems from collapsing into unmanageable template tangles.

3.5 Policies Are Not Configuration Objects

A common anti-pattern is treating policies as runtime configuration.
For example:

- Passing flags
- Using enums
- Reading settings at runtime

While sometimes necessary, this approach:

- Weakens static guarantees
- Increases testing complexity
- Pushes errors to runtime

Policies encode decisions that are:

- Stable
- Architectural
- Known at design time

If a decision must change dynamically, it is not a good candidate for a policy.

3.6 Policies as Building Blocks

A policy is rarely useful in isolation.
The real power emerges when policies are:

- Combined
- Reused
- Swapped independently

For example, a container might accept:

- A storage policy
- A bounds-checking policy
- A threading policy

Each policy contributes a narrow aspect of behavior, and the resulting type expresses the full design intent without inheritance hierarchies or runtime switches.

3.7 What Policies Are Not

To avoid misuse, it is important to be explicit.
A policy is **not**:

- A base class
- A configuration struct with data members
- A dumping ground for unrelated logic
- A template metaprogramming exercise

When policies become large or stateful, they stop being policies and start becoming subsystems.

3.8 Looking Ahead

This chapter established the conceptual foundation:

- Policies are types
- They encode behavioral decisions
- They are selected at compile time
- They must remain orthogonal and focused

The next chapter introduces *type policies and traits*, showing how types themselves can be used as compile-time information to guide behavior without introducing runtime cost.

Chapter 4

Type Policies and Traits

4.1 Why Types Can Carry Meaning

In C++, types are more than containers for data. They can encode intent, constraints, and decisions.
Policy-Based Design exploits this property by using types as compile-time signals that guide behavior. These signals are commonly referred to as *type policies*.
A type policy does not execute behavior directly. Instead, it communicates information that influences how behavior is selected or composed.
This chapter introduces type policies and traits, which together form the descriptive layer of policy-based systems.

4.2 Type Policies as Compile-Time Decisions

A type policy answers questions such as:

- Should bounds be checked?

- Is an operation allowed to throw?
- Is synchronization required?

These questions are architectural in nature. They should not depend on runtime conditions.
A simple example is a bounds-checking policy:

```
struct CheckedAccess {};
struct UncheckedAccess {};
```

These empty types already express a meaningful decision. Their purpose is not to store data, but to guide compilation.

4.3 Using Type Policies in Interfaces

Type policies are typically passed as template parameters:

```
template <typename AccessPolicy>
class Buffer;
```

Specialization or conditional logic can then adapt behavior:

```
template <>
class Buffer<CheckedAccess> {
public:
    int get(std::size_t index) const {
        if (index >= size_) {
            throw std::out_of_range("index");
        }
        return data_[index];
    }
private:
    int* data_;
    std::size_t size_;
};
```

```
template <>
class Buffer<UncheckedAccess> {
public:
    int get(std::size_t index) const noexcept {
        return data_[index];
    }
private:
    int* data_;
    std::size_t size_;
};
```

The interface remains uniform, while the behavior is statically selected.

4.4 Traits as Compile-Time Queries

Traits are a mechanism for querying properties of types at compile time.
The standard library uses traits extensively:

- `std::is_trivial`
- `std::is_copyable`
- `std::is_same`

A trait typically maps a type to a boolean or another type.
A simple custom trait might look like:

```
template <typename T>
struct is_checked_access : std::false_type {};
```

```
template <>
struct is_checked_access<CheckedAccess> : std::true_type {};
```

This allows code to branch at compile time without relying on specialization alone.

4.5 Conditional Compilation with Traits

Traits enable fine-grained conditional logic using `if constexpr` or type selection utilities.

```
template <typename AccessPolicy>
class Buffer {
public:
    int get(std::size_t index) const {
        if constexpr (is_checked_access<AccessPolicy>::value) {
            if (index >= size_) {
                throw std::out_of_range("index");
            }
        }
        return data_[index];
    }
private:
    int* data_;
    std::size_t size_;
};
```

This approach offers several advantages:

- No runtime branching
- No code duplication
- Clear expression of intent

The compiler removes unused paths entirely.

4.6 Type Policies vs Behavioral Policies

It is important to distinguish between:

- Type policies
- Behavioral policies

Type policies:

- Describe properties
- Influence decisions
- Often empty or stateless

Behavioral policies:

- Provide functions
- Implement actions
- May contain logic

Both are policies, but they serve different roles. Confusing them leads to bloated or unclear designs.

4.7 Tag Types and Intent Encoding

Tag types are a minimal form of type policy. They contain no members and exist purely to convey meaning.
Examples include:

- `std::in_place_t`
- `std::defer_lock_t`
- `std::try_to_lock_t`

These tags make interfaces:

- Self-documenting
- Unambiguous
- Resistant to misuse

Policy-Based Design generalizes this idea from function calls to entire class designs.

4.8 Scaling with Multiple Type Policies

Real systems often require multiple independent type policies.
For example:

```
template <
    typename AccessPolicy,
    typename ThreadPolicy,
    typename ErrorPolicy
>
class Container;
```

Each policy answers a distinct question. Together, they define the container's semantics.
The challenge is not technical, but architectural: choosing the right number of policies and keeping them orthogonal.

4.9 Guidelines for Designing Type Policies

Effective type policies follow simple rules:

- Represent one decision

- Avoid state
- Use expressive names
- Remain composable

When a type policy starts accumulating data or logic, it is often a sign that it should become a behavioral policy instead.

4.10 Preparing for Behavioral Policies

Type policies and traits provide the descriptive layer of Policy-Based Design.
They answer *what* decisions are in effect.
The next chapter introduces *behavioral policies*, which answer *how* those decisions are implemented, and shows how behavior itself can be injected cleanly and safely at compile time.

Chapter 5

Behavioral Policies

5.1 From Description to Action

In the previous chapter, type policies and traits were introduced as mechanisms for describing design decisions. They answer questions such as:

- Is this operation checked?
- Is this component thread-safe?
- Are exceptions permitted?

Behavioral policies complete the picture.
A *behavioral policy* is responsible for implementing a specific action or algorithm, while remaining configurable and interchangeable at compile time.
Where type policies describe intent, behavioral policies *execute that intent*.

5.2 Defining a Behavioral Policy

A behavioral policy is typically a class or struct that provides:

- One or more member functions
- A well-defined semantic responsibility
- No hidden global dependencies

Unlike type policies, behavioral policies often contain logic. They may also contain state, though this should be approached carefully.
A minimal example is a logging policy:

```
struct NullLogPolicy {
    static void log(const char*) noexcept {}
};
```

```
#include <iostream>

struct ConsoleLogPolicy {
    static void log(const char* message) {
        std::cout << message << '\n';
    }
};
```

Both policies expose the same interface, but their behavior differs radically.

5.3 Injecting Behavior via Templates

Behavioral policies are most commonly injected as template parameters.

```
template <typename LogPolicy>
class Service {
public:
    void run() {
        LogPolicy::log("Service started");
        // core logic
        LogPolicy::log("Service finished");
    }
};
```

At the call site, behavior is selected explicitly:

```
using SilentService  = Service<NullLogPolicy>;
using VerboseService = Service<ConsoleLogPolicy>;
```

No runtime checks. No conditionals. No virtual functions.
The selected behavior is fully visible to the compiler.

5.4 Stateless vs Stateful Behavioral Policies

Behavioral policies may be stateless or stateful.

5.4.1 Stateless Policies

Stateless policies:

- Use only static member functions
- Carry no data
- Are trivially optimizable

They are ideal for:

- Logging
- Validation
- Error handling strategies

Stateless policies are the safest and most common choice.

5.4.2 Stateful Policies

Sometimes, behavior requires state. For example, a logging policy might write to a file and need to maintain a file handle.

```
#include <fstream>

class FileLogPolicy {
public:
    explicit FileLogPolicy(const char* filename)
        : file_(filename) {}

    void log(const char* message) {
        file_ << message << '\n';
    }

private:
    std::ofstream file_;
};
```

Stateful policies introduce new considerations:

- Object lifetime
- Ownership
- Copy and move semantics

They can still be used effectively, but they require more discipline.

5.5 Passing Stateful Policies Safely

When policies are stateful, they are typically stored as members:

```
template <typename LogPolicy>
class Service {
public:
    explicit Service(LogPolicy log)
        : log_(std::move(log)) {}

    void run() {
        log_.log("Running service");
    }

private:
    LogPolicy log_;
};
```

This design:

- Makes ownership explicit
- Avoids hidden globals
- Preserves testability

The policy remains a first-class object, not an implicit dependency.

5.6 Behavioral Policies and Inlining

One of the key benefits of behavioral policies is that their calls are fully visible at compile time. When a policy function is:

- Known
- Small
- Stateless

The compiler will often inline it completely.
This means that policy-based code can achieve:

- Zero abstraction penalty
- Clean separation of concerns
- High performance

This is a critical distinction from runtime polymorphism, where inlining across virtual calls is limited.

5.7 Combining Type and Behavioral Policies

Type policies and behavioral policies are often used together.
For example:

```
template <
    typename AccessPolicy,
    typename LogPolicy
>
class Component {
public:
    void process(std::size_t index) {
        if constexpr (is_checked_access<AccessPolicy>::value) {
            // perform bounds check
        }
```

```
        LogPolicy::log("Processing element");
    }
};
```

In this design:

- Type policies guide decisions
- Behavioral policies perform actions
- Both remain independent and composable

This separation keeps each policy focused and reusable.

5.8 Avoiding Policy Overreach

A common mistake is allowing a behavioral policy to grow beyond its responsibility. Warning signs include:

- Multiple unrelated functions
- Hidden assumptions about the host class
- Direct access to host internals

A behavioral policy should be:

- Narrow
- Explicit
- Replaceable

If replacing a policy breaks unrelated code, the policy boundary is likely wrong.

5.9 Testing Behavioral Policies

Behavioral policies are naturally testable.
Because they are:

- Isolated
- Stateless or explicitly stateful
- Free of inheritance hierarchies

They can be tested independently from the components that use them.
In later chapters, we will explore how policy-based design improves unit testing and reduces mocking complexity.

5.10 Looking Forward

This chapter demonstrated how behavior itself can be injected cleanly and safely at compile time.
So far, we have:

- Defined policies
- Distinguished type and behavioral policies
- Injected behavior without runtime cost

The next chapter focuses on a particularly important category of behavioral policy: *storage and allocation policies*.
Memory management is where policy-based design often delivers its most dramatic benefits.

Chapter 6

Storage and Allocation Policies

6.1 Why Memory Strategy Is an Architectural Decision

Memory management is not an implementation detail. In C++, it is a fundamental architectural choice that affects:

- Performance
- Determinism
- Cache locality
- Exception safety
- Portability

Traditional designs often hard-code memory behavior:

- Using `new` and `delete` directly
- Relying on default allocators

- Embedding allocation logic inside core classes

This approach couples *what* a component does with *how* it manages memory.
Policy-Based Design separates these concerns.

6.2 What Is a Storage Policy?

A *storage policy* defines:

- Where objects are stored
- How memory is acquired
- How memory is released
- Whether allocation is dynamic or static

It does not define:

- Object semantics
- Business logic
- Algorithmic behavior

By isolating storage decisions, systems become easier to adapt to different environments without rewriting core logic.

6.3 A Simple Dynamic Allocation Policy

Consider a minimal dynamic allocation policy:

```
#include <memory>

template <typename T>
struct HeapStorage {
    static T* allocate() {
        return new T();
    }

    static void deallocate(T* ptr) noexcept {
        delete ptr;
    }
};
```

This policy:

- Allocates from the heap
- Uses default global operators
- Makes allocation explicit

While simple, it already isolates memory strategy from the class that uses it.

6.4 A Static Storage Policy

In embedded or performance-critical systems, dynamic allocation may be undesirable or forbidden.
A static storage policy can provide fixed storage:

```
template <typename T>
struct StaticStorage {
    alignas(T) static inline unsigned char buffer[sizeof(T)];
```

```
    static T* allocate() {
        return new (buffer) T();
    }

    static void deallocate(T* ptr) noexcept {
        ptr->~T();
    }
};
```

This policy:

- Avoids heap allocation
- Uses placement new
- Provides deterministic behavior

The consuming class does not need to change.

6.5 Using a Storage Policy in a Class

A class can be parameterized on its storage policy:

```
template <
    typename T,
    template <typename> class StoragePolicy
>
class Holder {
public:
    Holder() : ptr_(StoragePolicy<T>::allocate()) {}

    ~Holder() {
        StoragePolicy<T>::deallocate(ptr_);
```

```
    }

    T& get() { return *ptr_; }
    const T& get() const { return *ptr_; }

private:
    T* ptr_;
};
```

The memory strategy is now:

- Explicit
- Replaceable
- Enforced at compile time

6.6 Allocator-Based Storage Policies

The C++ standard library introduces allocators to generalize memory management. Policy-Based Design aligns naturally with this concept.

```
#include <memory>

template <typename T, typename Allocator = std::allocator<T>>
class AllocatorStorage {
public:
    explicit AllocatorStorage(const Allocator& alloc = Allocator{})
        : allocator_(alloc),
          ptr_(allocator_.allocate(1)) {
        std::allocator_traits<Allocator>::construct(allocator_, ptr_);
    }
```

```
    ~AllocatorStorage() {
        std::allocator_traits<Allocator>::destroy(allocator_, ptr_);
        allocator_.deallocate(ptr_, 1);
    }

    T& get() { return *ptr_; }

private:
    Allocator allocator_;
    T* ptr_;
};
```

This policy integrates cleanly with:

- Custom allocators
- Memory pools
- Tracking allocators

6.7 Storage Policies and Exception Safety

By isolating allocation logic, exception safety becomes easier to reason about.
Storage policies can:

- Guarantee no-throw allocation
- Centralize cleanup logic
- Enforce construction order

For example, a policy may document and enforce:

- Strong exception guarantees

- No-throw destruction
- Allocation failure strategies

This clarity is difficult to achieve when allocation is scattered across constructors.

6.8 Cache Locality and Embedded Storage

Storage policies can also optimize memory layout.
For small objects, embedding storage directly inside the owning object can improve cache locality:

```
template <typename T>
struct EmbeddedStorage {
    alignas(T) unsigned char buffer[sizeof(T)];

    T* allocate() {
        return new (buffer) T();
    }

    void deallocate(T* ptr) noexcept {
        ptr->~T();
    }
};
```

This avoids pointer indirection entirely, at the cost of increased object size.
Policy-Based Design allows this trade-off to be expressed explicitly.

6.9 Avoiding Over-Engineering

Storage policies are powerful, but they should not be introduced prematurely.
Signs that a storage policy is justified:

- Multiple allocation strategies are required
- The environment imposes constraints
- Performance or determinism is critical

If all instances use the same strategy and no variation is expected, a policy may add unnecessary complexity.

6.10 Key Design Guidelines

Effective storage policies follow these rules:

- Do one thing
- Make ownership explicit
- Document exception behavior
- Avoid hidden global state

When storage logic becomes entangled with business logic, the policy boundary is likely incorrect.

6.11 Looking Ahead

This chapter demonstrated how memory management can be lifted out of core classes and expressed as policies.
Next, we examine another critical dimension of behavior: *error handling policies*.
Error handling decisions often define the usability and reliability of a system, and policy-based design provides a disciplined way to express those decisions clearly and safely.

Chapter 7

Error Handling Policies

7.1 Error Handling as a Design Choice

Error handling is one of the most contentious aspects of C++ design. Different systems impose different requirements:

- Some rely heavily on exceptions
- Others forbid exceptions entirely
- Some require explicit error codes
- Others demand fail-fast behavior

A common mistake is embedding a single error-handling strategy deep inside core logic. This couples the system to one philosophy and makes reuse difficult.
Policy-Based Design treats error handling as a *first-class policy*, allowing the same logic to operate under different failure models.

7.2 What Is an Error Handling Policy?

An error handling policy defines:

- How errors are reported
- Whether failures throw, return, or terminate
- What guarantees are provided to callers

It does not define:

- When errors occur
- What constitutes an error
- How recovery is implemented by the caller

This separation keeps core logic focused while making failure semantics explicit.

7.3 Exception-Based Error Policy

An exception-based policy is suitable for systems where:

- Stack unwinding is acceptable
- Errors are exceptional
- Readability is prioritized

A simple exception policy might look like:

```cpp
#include <stdexcept>

struct ExceptionErrorPolicy {
    [[noreturn]]
    static void on_error(const char* message) {
        throw std::runtime_error(message);
    }
};
```

This policy clearly states:

- Errors do not return
- Callers must handle exceptions
- Failure propagates automatically

7.4 Error-Code-Based Policy

In low-level or embedded environments, exceptions may be prohibited.
An error-code-based policy returns explicit status values:

```cpp
#include <optional>

struct ErrorCodePolicy {
    static std::optional<const char*> on_error(const char* message)
    ↪  noexcept {
        return message;
    }
};
```

Here:

- Failure is explicit

- Control flow remains local
- No stack unwinding occurs

The calling code decides how to react.

7.5 Fail-Fast Policy

Some systems cannot tolerate recovery. In such cases, failing fast is preferable to continuing in an undefined state.

```
#include <cstdlib>

struct TerminatePolicy {
    [[noreturn]]
    static void on_error(const char* message) noexcept {
        std::abort();
    }
};
```

Fail-fast policies are common in:

- Safety-critical systems
- Internal invariants
- Debug configurations

Policy-Based Design allows this behavior to be selected explicitly and locally.

7.6 Integrating Error Policies into Components

An error policy is typically injected as a template parameter:

```
template <typename ErrorPolicy>
class Parser {
public:
    void parse(const char* input) {
        if (!input) {
            ErrorPolicy::on_error("Null input");
            return;
        }
        // parsing logic
    }
};
```

The core logic does not change. Only the error semantics vary.

7.7 Error Policies and `noexcept`

Error handling policies interact closely with exception specifications.
For example:

- An exception-based policy cannot be used in `noexcept` code
- A terminate policy can
- An error-code policy usually can

Policies make these constraints explicit.
A component can even enforce them:

```
static_assert(noexcept(ErrorPolicy::on_error("")),
              "Error policy must be noexcept");
```

Such checks prevent invalid configurations at compile time.

7.8 Combining Error Policies with Other Policies

Error handling rarely exists in isolation.
A component may combine:

- Storage policy
- Logging policy
- Error handling policy

For example:

```
template <
    typename StoragePolicy,
    typename LogPolicy,
    typename ErrorPolicy
>
class Engine;
```

Each policy remains focused, and the resulting type clearly communicates how failures are handled.

7.9 Testing Error Policies

Error policies are particularly easy to test.
They can be:

- Invoked directly
- Replaced with test doubles
- Verified independently

A test policy might record failures instead of throwing or terminating, allowing assertions without disrupting test flow.

7.10 Design Guidelines

Effective error handling policies:

- Make failure semantics obvious
- Avoid hidden side effects
- Respect system constraints
- Remain simple

If error handling logic becomes complex, it likely belongs outside the policy itself.

7.11 Looking Ahead

This chapter showed how error handling, often a source of architectural rigidity, can be cleanly isolated and expressed as a policy.
Next, we explore how multiple policies can be composed together safely, and how to manage interactions and ordering without sacrificing clarity.
The next chapter focuses on *policy composition*.

Chapter 8

Policy Composition

8.1 Why Composition Is the Real Challenge

Defining individual policies is relatively straightforward. The real difficulty in Policy-Based Design lies in *composition*: combining multiple independent policies into a coherent, maintainable whole.
As systems grow, components often need to vary along several dimensions:

- Memory management
- Error handling
- Logging
- Thread safety
- Validation

If policy composition is not handled carefully, the design can quickly become unreadable or fragile. This chapter focuses on how to compose policies cleanly and safely.

8.2 Independent Dimensions of Behavior

The first principle of policy composition is independence.
Each policy should:

- Address a single concern
- Be unaware of other policies
- Avoid shared assumptions

For example:

```
template <
    typename StoragePolicy,
    typename LogPolicy,
    typename ErrorPolicy
>
class Component;
```

Each policy answers a different question:

- How is memory managed?
- How is information logged?
- How are errors reported?

When policies remain independent, they can be mixed and matched freely.

8.3 Layered Policy Application

One effective approach to composition is to apply policies in layers.
Consider a component method:

```
void process();
```

Internally, behavior may follow this conceptual order:

- Validate inputs
- Perform core logic
- Log results
- Handle errors

Each layer can correspond to a policy, without any policy knowing about the others explicitly.

8.4 Delegation Through Policy Interfaces

Policies should interact with the host class through minimal, well-defined interfaces.
For example, a logging policy should not access internal data structures directly.
Instead:

```
template <typename LogPolicy>
class Worker {
public:
    void execute() {
        LogPolicy::log("Starting work");
        do_work();
        LogPolicy::log("Work completed");
    }
private:
    void do_work();
};
```

This approach:

- Preserves encapsulation
- Prevents hidden coupling
- Simplifies reasoning

8.5 Ordering and Dependency Constraints

Not all policies are entirely independent. Sometimes, ordering matters.
For example:

- Error logging should occur before termination
- Validation should occur before allocation

Rather than encoding these constraints implicitly, they should be made explicit.
One approach is to define a composite policy:

```
template <typename LogPolicy, typename ErrorPolicy>
struct LoggedErrorPolicy {
    [[noreturn]]
    static void on_error(const char* message) {
        LogPolicy::log(message);
        ErrorPolicy::on_error(message);
    }
};
```

This composite policy:

- Expresses ordering explicitly
- Avoids entangling unrelated policies
- Remains reusable

8.6 Avoiding Policy Interdependence

A dangerous anti-pattern is allowing one policy to depend directly on another.
For example:

- A storage policy calling a logging policy
- An error policy assuming a threading model

Such dependencies:

- Break orthogonality
- Reduce reuse
- Create fragile designs

If two policies must interact, their interaction should be lifted into a higher-level policy, not embedded in either one.

8.7 Managing Template Parameter Explosion

As policies accumulate, template parameter lists can become long.

```
template <
    typename StoragePolicy,
    typename AccessPolicy,
    typename ThreadPolicy,
    typename LogPolicy,
    typename ErrorPolicy
>
class Engine;
```

This is not inherently wrong, but it can harm readability.
Common mitigation strategies include:

- Policy bundles
- Default policies
- Named type aliases

8.8 Policy Bundles

A policy bundle groups related policies into a single type.

```
struct DefaultPolicies {
    using Storage = HeapStorage;
    using Access  = UncheckedAccess;
    using Log     = NullLogPolicy;
    using Error   = ExceptionErrorPolicy;
};
```

The component then depends on the bundle:

```
template <typename Policies>
class Engine;
```

This approach:

- Reduces parameter noise
- Encourages consistent configurations
- Simplifies usage

8.9 Defaults and Partial Customization

Policies should usually have sensible defaults.

```
template <
    typename StoragePolicy = HeapStorage,
    typename ErrorPolicy   = ExceptionErrorPolicy
>
class Manager;
```

Defaults allow:

- Simple usage for common cases
- Advanced customization when needed
- Gradual adoption of policies

Users override only what matters.

8.10 Compile-Time Validation of Policy Sets

Policy-Based Design allows invalid combinations to be rejected at compile time.
For example:

```
static_assert(
    !std::is_same_v<ThreadSafePolicy, SingleThreadPolicy>,
    "Conflicting thread policies"
);
```

Such checks:

- Catch errors early

- Document assumptions
- Improve diagnostics

Validation logic should be simple and local, not scattered across the codebase.

8.11 Balancing Power and Clarity

Policy composition is powerful, but it must be guided by discipline.
Warning signs include:

- Deeply nested templates
- Cryptic type names
- Excessive specialization

The goal is not maximal flexibility, but controlled, explicit variability.

8.12 Looking Ahead

This chapter showed how policies can be combined without sacrificing clarity or safety.
So far, we have covered:

- Type policies
- Behavioral policies
- Storage policies
- Error handling policies
- Composition techniques

Next, we compare Policy-Based Design with classical object-oriented approaches, and examine when each model is appropriate.

The next chapter focuses on *Policy-Based Design vs Object-Oriented Design*.

Chapter 9

Policy-Based Design vs Object-Oriented Design

9.1 Two Design Philosophies, Not Opponents

Policy-Based Design and Object-Oriented Design are often presented as competing paradigms. This framing is misleading.
They are not adversaries. They are responses to different kinds of problems.
Object-Oriented Design (OOD) emphasizes:

- Runtime polymorphism
- Substitutability
- Encapsulation through inheritance

Policy-Based Design emphasizes:

- Compile-time composition

- Static guarantees
- Explicit architectural decisions

Understanding when to apply each approach is far more important than choosing one exclusively.

9.2 Strengths of Classical Object-Oriented Design

Object-Oriented Design excels when:

- Behavior must change at runtime
- Interfaces must remain stable across binaries
- Systems require plugin-style extensibility

Common examples include:

- GUI frameworks
- Device drivers
- Application-level extension systems

In these domains, runtime polymorphism is not a liability. It is a requirement.

9.3 Limitations of Object-Oriented Design in C++

While OOD is powerful, its limitations become apparent in performance-critical or library-level code.
Common issues include:

- Rigid inheritance hierarchies
- Hidden coupling through base classes
- Difficulty expressing orthogonal variation

As systems evolve, inheritance trees often become:

- Deep
- Fragile
- Resistant to refactoring

These issues are not failures of OOP itself, but of applying it beyond its natural scope.

9.4 Where Policy-Based Design Excels

Policy-Based Design is particularly effective when:

- Behavior is known at compile time
- Performance and inlining matter
- Libraries must support many configurations

Typical domains include:

- Containers and algorithms
- Infrastructure libraries
- Embedded and systems software

In these contexts, runtime polymorphism often introduces unnecessary cost and complexity.

9.5 Expressiveness vs Flexibility

One of the key trade-offs between the two approaches is expressiveness versus flexibility. Object-Oriented Design offers:

- Runtime flexibility
- Late binding
- Dynamic substitution

Policy-Based Design offers:

- Compile-time expressiveness
- Strong static guarantees
- Precise control over behavior

Choosing the wrong model leads to:

- Over-engineering
- Performance penalties
- Unnecessary abstraction

9.6 Inheritance vs Composition by Types

Inheritance couples behavior and structure. Policy-Based Design decouples them. Consider inheritance-based design:

- Behavior is acquired implicitly

- Changes ripple through the hierarchy
- Base classes accumulate responsibilities

In contrast, policy-based composition:

- Behavior is explicit
- Changes are localized
- Responsibilities remain isolated

This distinction becomes critical as systems grow and requirements evolve.

9.7 Binary Interfaces and ABI Stability

One area where Object-Oriented Design retains an advantage is binary interface stability. Virtual interfaces:

- Enable ABI-compatible extensions
- Support plugin architectures
- Decouple compilation units

Policy-Based Design, being template-based:

- Requires recompilation
- Is not ABI-stable by default
- Operates primarily at the source level

This makes policy-based techniques more suitable for header-only libraries and internal systems.

9.8 Testing and Reasoning About Code

Policy-Based Design improves testability by isolating behavior into replaceable units.
Compared to inheritance-based designs:

- No need for complex mock hierarchies
- Fewer hidden dependencies
- Clearer ownership and lifetimes

Policies can be:

- Tested independently
- Substituted explicitly
- Validated at compile time

This leads to simpler and more robust test suites.

9.9 Hybrid Designs: Using Both Effectively

Most real-world systems benefit from a hybrid approach.
A common pattern is:

- Policy-Based Design for core logic
- Object-Oriented Design for system boundaries

For example:

- A policy-based engine behind a virtual interface
- Static composition internally, dynamic selection externally

This combination leverages the strengths of both paradigms without inheriting their weaknesses.

9.10 Design Smells and Warning Signs

Indicators that Object-Oriented Design is being overused:

- Deep inheritance trees
- Frequent dynamic casts
- Base classes with unrelated responsibilities

Indicators that Policy-Based Design is being overused:

- Excessive template parameters
- Hard-to-read compiler errors
- Compile times dominating development

Good design requires balance and judgment.

9.11 Choosing the Right Tool

There is no universal answer.
A simple guideline is:

- If behavior must vary at runtime, prefer OOD
- If behavior is fixed and performance matters, prefer policies

Policy-Based Design is not a replacement for Object-Oriented Design. It is a refinement for cases where C++'s compile-time capabilities offer a clearer, safer, and more efficient solution.

9.12 Looking Ahead

With the conceptual comparison complete, we now turn to modern language features that significantly improve the usability of Policy-Based Design.
The next chapter explores how *Modern C++ (C++20/23)* simplifies constraints, improves diagnostics, and makes policy-based systems more readable and maintainable than ever before.

Chapter 10

Policy-Based Design in Modern C++ (C++20/23)

10.1 Why Modern C++ Changes the Equation

Policy-Based Design existed long before C++20. Early implementations relied on:

- Template specialization
- SFINAE
- Cryptic compiler diagnostics

While powerful, these techniques made policy-based code:

- Hard to read
- Hard to teach
- Hard to debug

Modern C++ fundamentally changes this landscape.
With the introduction of concepts, `constexpr` evolution, and improved type traits,
Policy-Based Design becomes:

- Safer
- More expressive
- Significantly more readable

This chapter shows how C++20 and C++23 elevate policy-based systems from expert-only techniques to practical, maintainable designs.

10.2 Concepts as Explicit Policy Contracts

Before C++20, policy requirements were implicit. Violations resulted in long, unreadable error messages.
Concepts allow policies to be expressed as *named contracts*.

```
template <typename P>
concept LogPolicy =
requires(const char* msg) {
    { P::log(msg) };
};
```

This concept states clearly:

- What a policy must provide
- What operations are required
- What signatures are expected

The consuming code becomes self-documenting.

10.3 Constraining Policy Parameters

Concepts can be used directly in templates:

```
template <LogPolicy Logger>
class Service {
public:
    void run() {
        Logger::log("Service running");
    }
};
```

If an invalid policy is provided, the compiler emits a clear diagnostic.

This eliminates an entire class of misuse that previously required documentation and discipline.

10.4 Replacing SFINAE with `requires`

SFINAE-based designs often obscure intent.

Modern C++ allows constraints to be written explicitly:

```
template <typename P>
requires requires { P::allocate(); }
void use_storage(P& policy) {
    // use allocation
}
```

This approach:

- Improves readability
- Localizes constraints
- Produces better error messages

Policy-Based Design benefits enormously from this clarity.

10.5 `if constexpr` and Policy Logic

Conditional behavior based on policies is far cleaner with `if constexpr`.

```
template <typename ErrorPolicy>
void process(bool ok) {
    if (!ok) {
        if constexpr (requires {
            ErrorPolicy::on_error("");
        }) {
            ErrorPolicy::on_error("Failure");
        }
    }
}
```

The compiler:

- Evaluates conditions at compile time
- Removes unreachable branches
- Avoids runtime overhead

This makes policy-driven logic straightforward and maintainable.

10.6 Stronger `constexpr` Capabilities

Modern C++ significantly expands what can be evaluated at compile time.
Policies can now:

- Contain `constexpr` logic
- Perform compile-time validation

- Influence data layout decisions

Example:

```
struct CheckedPolicy {
    static constexpr bool enabled = true;
};
```

```
template <typename Policy>
constexpr int compute(int value) {
    if constexpr (Policy::enabled) {
        return value > 0 ? value : 0;
    } else {
        return value;
    }
}
```

This code generates optimal machine instructions for each policy configuration.

10.7 Improved Diagnostics and Maintainability

One of the strongest arguments for modern policy-based design is diagnostic quality. With concepts:

- Errors point to violated requirements
- Messages reference domain terms
- Debugging time is reduced

This transforms policy-based code from an expert-only technique into a maintainable architectural tool.

10.8 Reducing Template Noise

Modern C++ encourages clearer interfaces.
Techniques include:

- Named concepts
- Policy bundles
- Default template arguments

Example:

```
template <
    typename Policies = DefaultPolicies
>
class Engine;
```

Clarity improves without sacrificing flexibility.

10.9 Constexpr-Friendly Policy Validation

Policies can validate themselves at compile time.

```
template <typename P>
consteval void validate_policy() {
    static_assert(P::enabled || !P::enabled,
                  "Invalid policy state");
}
```

Such checks:

- Catch misconfiguration early
- Encode design assumptions
- Serve as living documentation

10.10 Modern C++ Enables Safer Abstractions

The historical criticism of policy-based design focused on complexity and usability. Modern C++ addresses these concerns directly:

- Concepts replace SFINAE
- `constexpr` replaces runtime logic
- Diagnostics improve developer experience

The result is not more abstraction, but better abstraction.

10.11 Design Guidelines for Modern Policy-Based Code

When using modern C++ features:

- Prefer concepts over documentation
- Prefer `if constexpr` over specialization
- Keep policies small and explicit
- Validate aggressively at compile time

These guidelines keep policy-based systems approachable and robust.

10.12 Looking Ahead

With modern language support in place, we can now focus on the practical benefits of policy-based design in large systems.

The next chapter examines *testability and maintainability*, showing how policy-based architectures simplify testing, reduce mocking, and improve long-term evolution of C++ codebases.

Chapter 11

Testability and Maintainability

11.1 Why Architecture Determines Testability

In large C++ systems, testability is rarely a property that can be added after the fact. It emerges directly from architectural decisions.
Code becomes difficult to test when:

- Dependencies are implicit
- Behavior is selected at runtime through hidden mechanisms
- Side effects are scattered across the system

Policy-Based Design addresses these issues at their root by making dependencies explicit and behavior selectable at compile time.
This chapter explains why policy-based architectures naturally lead to code that is easier to test, reason about, and evolve over time.

11.2 Explicit Dependencies by Construction

One of the strongest benefits of Policy-Based Design is that dependencies are expressed directly in types.
When a class is parameterized by policies:

```
template <
    typename StoragePolicy,
    typename LogPolicy,
    typename ErrorPolicy
>
class Engine;
```

Every dependency is:

- Visible at the interface
- Explicit in the type system
- Impossible to forget or bypass

This contrasts with traditional designs, where dependencies may be:

- Hidden behind base classes
- Injected through globals or singletons
- Selected indirectly through configuration

Explicit dependencies make reasoning and testing far simpler.

11.3 Testing Policies in Isolation

Policies are, by design, small and focused. This makes them ideal candidates for isolated testing. A behavioral policy can often be tested without involving the component that uses it.
Example:

```
struct RecordingLogPolicy {
    static inline int count = 0;

    static void log(const char*) {
        ++count;
    }
};
```

A test can verify behavior directly:

```
RecordingLogPolicy::count = 0;
RecordingLogPolicy::log("test");
assert(RecordingLogPolicy::count == 1);
```

No mocks. No inheritance hierarchies. No framework dependencies.

11.4 Mock Policies Instead of Mock Objects

Traditional object-oriented designs often rely on mock objects derived from virtual interfaces. This approach introduces:

- Additional inheritance layers
- Fragile test setups
- Tight coupling to interface details

Policy-Based Design replaces mock objects with mock policies.
A test-specific policy can be defined inline:

```
struct TestErrorPolicy {
    static inline bool error_called = false;

    static void on_error(const char*) noexcept {
        error_called = true;
    }
};
```

The component under test is instantiated with this policy explicitly.
This approach is:

- Simpler
- More explicit
- Less brittle

11.5 Compile-Time Validation Reduces Test Burden

Policies allow many classes of errors to be caught at compile time.
Examples include:

- Invalid policy combinations
- Missing required functions
- Incorrect exception guarantees

By enforcing constraints statically, the number of runtime tests required to verify configuration correctness is reduced.
Tests can focus on behavior, not structural validity.

11.6 Deterministic Behavior Simplifies Testing

Runtime polymorphism often introduces:

- Indirect calls
- Late binding
- Hidden state

These features complicate testing, especially when timing or ordering matters.
Policy-Based Design selects behavior at compile time. As a result:

- Call graphs are fixed
- Behavior is deterministic
- Side effects are localized

This determinism makes tests:

- Easier to write
- Easier to debug
- Easier to trust

11.7 Improved Refactorability

Well-designed policies isolate change.
When a requirement changes:

- A new policy can be introduced

- Existing policies remain untouched
- Core logic stays stable

This reduces the risk associated with refactoring. Tests written against existing policies continue to pass, providing confidence during evolution.

11.8 Reducing the Need for Integration Tests

Integration tests are expensive to write and maintain. They are often necessary when:

- Components are tightly coupled
- Behavior cannot be isolated
- Side effects span multiple subsystems

Policy-Based Design reduces this pressure.
Because behavior is composed from independent policies, many interactions can be tested at the unit level.
Integration tests still have value, but they no longer carry the entire testing burden.

11.9 Long-Term Maintainability

Maintainability is not just about readability. It is about the cost of change over time.
Policy-based systems tend to:

- Age gracefully
- Accumulate fewer workarounds
- Resist architectural decay

Because decisions are explicit and localized, new developers can understand:

- What varies
- What is fixed
- Where to make changes

This clarity is invaluable in long-lived systems.

11.10 Avoiding Test Fragility

Tests become fragile when they depend on:

- Internal structure
- Implicit behavior
- Global state

Policy-Based Design encourages:

- Narrow interfaces
- Explicit configuration
- Stateless components

As a result, tests are more resilient to internal refactoring.

11.11 Design Guidelines for Testable Policies

To maximize testability:

- Keep policies small
- Avoid hidden state
- Prefer stateless policies when possible
- Make side effects explicit

When state is required, ensure ownership and lifetime are clear.

11.12 Looking Ahead

At this point, we have covered:

- Core policy concepts
- Composition techniques
- Modern language support
- Testability benefits

The next chapter presents a concrete, end-to-end case study.
We will take an inheritance-heavy design and refactor it step by step into a clean, policy-based architecture, highlighting practical decisions and trade-offs.

Chapter 12

A Real-World Case Study: From Inheritance to Policies

12.1 Why a Case Study Matters

Policy-Based Design is often understood in theory but questioned in practice.
Developers ask:

- Is this really simpler?
- Does this scale in real code?
- What does refactoring actually look like?

This chapter answers those questions by walking through a realistic refactoring scenario: transforming a traditional inheritance-heavy design into a clean, policy-based architecture. The goal is not to present an idealized example, but a believable one that mirrors real-world constraints.

12.2 The Original Inheritance-Based Design

Consider a simplified data processing component used in multiple environments. Requirements include:

- Optional logging
- Different error handling strategies
- Multiple memory allocation models

A classical object-oriented design might begin like this:

```
class Processor {
public:
    virtual ~Processor() = default;
    virtual void process(int value) = 0;
};
```

Derived classes then multiply:

```
class LoggingProcessor : public Processor {
protected:
    void log(const char* msg) {
        // logging logic
    }
};
```

```
class ExceptionProcessor : public LoggingProcessor {
public:
    void process(int value) override {
        if (value < 0) {
            throw std::runtime_error("Invalid value");
        }
```

```
        log("Processing value");
        // work
    }
};
```

Soon, additional combinations appear:

- Non-logging + exception
- Logging + error codes
- No allocation + fail-fast

The hierarchy grows. Complexity follows.

12.3 Problems Observed in Practice

As the system evolves, several issues emerge:

- Inheritance hierarchies encode multiple concerns
- Adding a new behavior requires new subclasses
- Behavior combinations are implicit and fragile

Testing becomes difficult:

- Mocking requires inheritance
- Base classes leak assumptions
- Side effects are hard to isolate

The design works—but it does not scale.

12.4 Identifying Orthogonal Concerns

The first refactoring step is conceptual, not technical.
We identify independent dimensions of variation:

- Logging behavior
- Error handling behavior
- Storage strategy

Each concern will become a policy.
This step is critical. If concerns are misidentified, the resulting policy design will be flawed.

12.5 Defining the Policies

We begin by defining minimal policies.

12.5.1 Logging Policies

```
struct NullLogPolicy {
    static void log(const char*) noexcept {}
};
```

```
struct ConsoleLogPolicy {
    static void log(const char* msg) {
        std::cout << msg << '\n';
    }
};
```

12.5.2 Error Handling Policies

```
struct ExceptionPolicy {
    [[noreturn]]
    static void on_error(const char* msg) {
        throw std::runtime_error(msg);
    }
};
```

```
struct ErrorCodePolicy {
    static bool on_error(const char*) noexcept {
        return false;
    }
};
```

12.5.3 Storage Policy

```
template <typename T>
struct HeapStorage {
    static T* allocate() {
        return new T();
    }

    static void deallocate(T* ptr) noexcept {
        delete ptr;
    }
};
```

12.6 Rebuilding the Processor with Policies

We now express the processor in terms of policies:

```
template <
```

```
    typename LogPolicy,
    typename ErrorPolicy,
    template <typename> class StoragePolicy
>
class Processor {
public:
    void process(int value) {
        if (value < 0) {
            ErrorPolicy::on_error("Invalid value");
            return;
        }

        LogPolicy::log("Processing value");
        // work
    }
};
```

Notice what has changed:

- No inheritance hierarchy
- No virtual functions
- No implicit behavior

All behavior is explicit in the type.

12.7 Instantiating Concrete Configurations

Concrete configurations are easy to express:

```
using SafeProcessor =
    Processor<ConsoleLogPolicy, ExceptionPolicy, HeapStorage>;
```

```
using SilentProcessor =
    Processor<NullLogPolicy, ErrorCodePolicy, HeapStorage>;
```

Each alias:

- Documents intent
- Defines behavior unambiguously
- Introduces no runtime cost

12.8 Testing Becomes Trivial

Testing no longer requires mocks or subclassing.

```
struct TestLogPolicy {
    static inline int calls = 0;
    static void log(const char*) {
        ++calls;
    }
};
```

```
using TestProcessor =
    Processor<TestLogPolicy, ErrorCodePolicy, HeapStorage>;
```

The test can now assert behavior directly, without touching internal logic.

12.9 Performance and Clarity Gains

After refactoring:

- Virtual dispatch is eliminated

- Logging calls may be inlined
- Dead code is removed by the compiler

Equally important, the architecture becomes clearer:

- Behavior is explicit
- Variation is controlled
- Change is localized

12.10 Lessons Learned

This case study highlights key insights:

- Policy-Based Design scales where inheritance does not
- Compile-time composition clarifies architecture
- Testing and refactoring become simpler

The refactoring did not reduce functionality. It reduced coupling.

12.11 Looking Ahead

With the core material complete, the remaining sections of this booklet focus on common mistakes, design checklists, and practical guidance for adoption.
The next chapter examines *common anti-patterns* encountered in policy-based systems and how to avoid them.

Appendices

Appendix A: Common Anti-Patterns in Policy-Based Design

Why Anti-Patterns Matter

Policy-Based Design is a powerful architectural technique, but like any powerful tool, it can be misused.
Many failed attempts at policy-based systems do not fail because the idea is flawed, but because certain recurring mistakes are made.
This appendix documents the most common anti-patterns observed in real C++ codebases, along with guidance on how to avoid them.
Understanding these pitfalls is essential for using policies responsibly and effectively.

Over-Templatization

One of the most frequent mistakes is introducing templates where no variability exists.
Symptoms include:

- Template parameters that are never changed
- Policies with only one valid implementation
- Excessive abstraction without clear benefit

This leads to:

- Increased compile times
- Reduced readability
- Unnecessary cognitive overhead

Guideline: Introduce a policy only when there is a genuine need for variation.

Policies That Do Too Much

A policy should represent a single decision or behavior.
An anti-pattern occurs when a policy:

- Handles multiple unrelated responsibilities
- Contains complex internal logic
- Becomes a miniature subsystem

Such policies:

- Are hard to reason about
- Are difficult to reuse
- Break orthogonality

Guideline: If a policy grows large, split it into multiple focused policies.

Hidden Coupling Between Policies

Policies should be independent.
A common mistake is allowing one policy to assume the presence or behavior of another.
Examples include:

- A logging policy that assumes exceptions are enabled
- A storage policy that assumes thread safety
- An error policy that writes to a specific log

This hidden coupling:

- Reduces composability
- Causes surprising failures
- Makes reuse difficult

Guideline: If two policies must interact, create a higher-level composite policy that explicitly encodes the relationship.

Encoding Runtime Decisions as Policies

Policies are intended for decisions that are stable and architectural.
An anti-pattern occurs when:

- Runtime configuration flags are turned into policies
- User-driven options are hard-coded into types
- Behavior that must change dynamically is fixed statically

This results in:

- Code duplication
- Excessive recompilation
- Reduced flexibility

Guideline: If a decision must change at runtime, it is not a good candidate for a policy.

Policy Leakage into Core Logic

Core components should not be aware of policy implementation details.
An anti-pattern is present when:

- The host class inspects policy internals
- Policy-specific logic spreads across the codebase
- Core algorithms depend on specific policies

This undermines the purpose of policies and reintroduces tight coupling.
Guideline: Interact with policies only through their defined interfaces or contracts.

Unreadable Template Interfaces

Long template parameter lists and cryptic type names can obscure intent.
For example:

- Deeply nested template aliases
- Abbreviated or unclear policy names
- Excessive use of specialization

This makes code:

- Hard to learn
- Difficult to debug
- Resistant to adoption

Guideline: Favor clarity over cleverness. Use descriptive names and policy bundles to keep interfaces readable.

Ignoring Compile-Time Diagnostics

Policy-based systems offer powerful compile-time validation, but this is often underused. Anti-patterns include:

- Missing static assertions
- Silent assumptions about policy behavior
- Allowing invalid configurations to compile

This shifts errors to runtime, defeating one of the main benefits of the approach.
Guideline: Validate aggressively at compile time. Fail early and loudly.

Premature Optimization Through Policies

Policies are sometimes introduced purely for perceived performance gains.
This can lead to:

- Overly complex designs
- Hard-to-maintain code
- Minimal real-world benefit

Guideline: Use policies to express architecture, not to micro-optimize prematurely. Measure before optimizing.

Summary of Anti-Patterns

Most policy-based design failures can be traced back to a small set of mistakes:

- Overuse of templates
- Poor separation of concerns
- Hidden dependencies
- Misaligned decision timing

Avoiding these pitfalls allows Policy-Based Design to remain a powerful, elegant, and sustainable technique.

Policy Design Checklist

Purpose of the Checklist

Designing good policies requires discipline. This checklist serves as a practical guide for evaluating whether a policy is well-designed and appropriate for use.
It can be applied during:

- Initial design
- Code reviews
- Refactoring efforts

Policy Scope and Responsibility

Before introducing a policy, ask:

- Does this policy represent a single decision?
- Is the responsibility clearly defined?
- Can the policy be explained in one sentence?

If the answer to any of these is no, the policy is likely too broad.

Compile-Time Suitability

Evaluate whether the decision belongs at compile time:

- Is the behavior stable across executions?
- Does it depend on user input?
- Is runtime flexibility required?

Policies should encode architectural decisions, not runtime variability.

Orthogonality and Composition

Check composability:

- Can this policy be combined with others freely?
- Does it assume the presence of another policy?
- Does it introduce ordering constraints?

Policies should remain independent whenever possible.

Interface Clarity

Review the policy interface:

- Are required functions obvious?
- Are semantics well-defined?
- Are names descriptive?

A policy interface should serve as documentation.

Testability

Assess testability:

- Can the policy be tested in isolation?
- Can it be replaced with a test policy?
- Are side effects explicit?

If testing is difficult, the policy boundary may be incorrect.

Complexity and Size

Evaluate complexity:

- Is the policy small?
- Is logic minimal?
- Does it avoid unnecessary state?

Large or complex policies often indicate missing abstraction boundaries.

Long-Term Maintainability

Finally, consider evolution:

- Will future changes affect this policy?
- Can new policies be added without modification?
- Is the design resilient to growth?

A good policy-based design anticipates change without becoming rigid.

Conclusion

Policy-Based Design as an Architectural Tool

Policy-Based Design is often misunderstood as a niche or overly complex technique.
In reality, it is a disciplined architectural approach that leverages C++'s strongest features:

- Compile-time composition
- Zero-overhead abstraction
- Strong static guarantees

When applied correctly, it enables systems that are:

- Flexible
- Performant
- Testable
- Maintainable

What Policy-Based Design Is Not

It is not:

- A replacement for Object-Oriented Design
- An excuse for template complexity
- A one-size-fits-all solution

Policy-Based Design is a tool, and like all tools, its value depends on how and when it is used.

A Mindset Shift

Adopting policy-based design requires a shift in thinking:

- From inheritance to composition
- From runtime decisions to compile-time intent
- From implicit behavior to explicit architecture

This shift aligns closely with the philosophy of modern C++.

Final Thoughts

Policy-Based Design rewards engineers who think carefully about boundaries, responsibilities, and long-term evolution.
It is most effective when:

- Used deliberately
- Applied incrementally
- Guided by clarity and restraint

When those conditions are met, it becomes one of the most powerful design techniques available to C++ programmers.

References

This section documents the primary sources, standards, and bodies of knowledge that informed the concepts, terminology, and design principles presented throughout this booklet.
The references are grouped by category and expanded with contextual explanations to clarify their relevance to Policy-Based Design in modern C++.

ISO C++ Core Guidelines

The ISO C++ Core Guidelines represent a living, continuously refined body of best practices for writing safe, modern, and maintainable C++ code.
They are particularly relevant to Policy-Based Design because they emphasize:

- Explicit ownership and lifetime management
- Clear separation of responsibilities
- Avoidance of hidden dependencies
- Zero-overhead abstractions

Many principles advocated by policy-based architectures—such as making design decisions explicit and enforcing them at compile time—align directly with the Core Guidelines' philosophy.
Key thematic relevance includes:

- Preference for compile-time checking over runtime failures
- Use of templates as architectural tools, not mere code generators
- Emphasis on clarity, readability, and long-term maintainability

While the guidelines are not prescriptive frameworks, they provide an essential conceptual foundation for disciplined policy design.

Andrei Alexandrescu — *Modern C++ Design*

Andrei Alexandrescu's *Modern C++ Design* is the seminal work that formally introduced Policy-Based Design to the C++ community.
The book pioneered:

- Policy classes as compile-time behavioral components
- Type-based configuration of class behavior
- Static composition as an alternative to deep inheritance

Many of the ideas explored in this booklet—such as orthogonal policy composition, storage policies, and error handling policies—trace their conceptual origin to this work.
However, this booklet deliberately modernizes those ideas by:

- Reframing them in the context of C++17–C++23
- Replacing SFINAE-heavy techniques with concepts and `requires`
- Emphasizing maintainability and diagnostics over metaprogramming cleverness

Alexandrescu's work remains foundational, but it should be read as a starting point, not as a final or static prescription.

ISO C++ Language Standards

The evolution of the C++ language itself is a critical reference for policy-based architectures. This booklet assumes familiarity with and selectively builds upon features introduced in:

C++17

C++17 stabilized many language features that made policy-based designs more practical and expressive, including:

- `if constexpr`
- Improved `constexpr` capabilities
- Structured bindings
- Refined type traits

These features significantly reduced the complexity of conditional compile-time logic.

C++20

C++20 marked a turning point for Policy-Based Design.
Its most important contribution is:

- Concepts and `requires` expressions

These features allow:

- Explicit expression of policy contracts
- Clear, readable constraints
- Dramatically improved compiler diagnostics

C++20 transforms policy-based code from an expert-only technique into a maintainable architectural approach.

C++23

C++23 further refines compile-time expressiveness and consistency, improving:

- `constexpr` usability
- Compile-time validation patterns
- Library support for modern design idioms

While no single feature defines policy-based design in C++23, the cumulative effect is a language that better supports explicit, type-driven architectures.

Compiler Documentation

Understanding compiler behavior is essential when working with policy-based systems, as such designs rely heavily on template instantiation, inlining, and optimization.
Relevant compiler families include:

GCC

GCC documentation provides insight into:

- Template instantiation models
- Optimization passes affecting inlining and dead-code elimination
- Diagnostic controls for template-heavy code

This knowledge helps developers reason about compile-time cost and generated machine code.

Clang / LLVM

Clang is particularly valuable for policy-based design due to:

- High-quality template diagnostics
- Clear error messages for concepts and constraints
- Strong support for modern C++ standards

LLVM's optimization pipeline also provides practical insight into how zero-overhead abstractions are realized in practice.

MSVC

MSVC documentation is essential for:

- Understanding Windows-specific ABI considerations
- Managing template instantiation across translation units
- Ensuring portability of policy-based libraries on Windows platforms

Policy-based designs intended for cross-platform use must account for compiler-specific behavior and limitations.

Large-Scale C++ System Case Studies

Beyond formal standards and books, many insights in this booklet are informed by patterns observed in large, long-lived C++ systems.
These systems typically exhibit:

- Heavy use of compile-time configuration

- Minimal reliance on runtime polymorphism in core logic
- Clear separation between architectural and dynamic behavior

Case studies from domains such as:

- Infrastructure libraries
- Game engines
- Embedded systems
- Financial and trading systems

demonstrate that policy-based techniques are not academic curiosities, but practical tools for managing complexity at scale.

Closing Note on References

This booklet intentionally avoids treating any single reference as authoritative in isolation. Policy-Based Design is best understood as:

- An architectural mindset
- Informed by language evolution
- Grounded in real-world system constraints

Readers are encouraged to study these references critically, compare approaches, and adapt techniques to the specific needs of their own systems.
Strong design emerges not from imitation, but from understanding.