The Roles of Symmetry And Orthogonality In Design

CHARLEY BAY
The Roles of Symmetry And Orthogonality In Design
"Either you keep self improving, or it's time to move into management."

--Niall Douglas
30-Sep-2021

https://old.reddit.com/r/cpp/comments/pye3iv/c_commi
ttee_dont_want_to_fix_rangebased_for_loop/heug4b/
Today’s Agenda

• Levels of “Knowing”
• Role of Symmetry
• Role of Asymmetry
• Role of Orthogonality
• Design Relationships
• Conclusion
Levels of “Knowing”

Understanding without tedious scrutiny
What’s the purpose of this?
• Q: What Does “Design” Provide?

• A:
Q: What Does “Design” Provide?

A: We “Know”: How the structure and behavior achieves a desired result
Q: What Does “Design” Provide?

A: We “Know”:
How the structure and behavior achieves a desired result

We understand the inner-workings of our system

Is “obvious” or “clear”

Our “First” or “Best Guess” to any question is usually correct
Levels of “Knowing”

Guarantee
- Inviolate principle or behavior

Examples:
- **C++ Language Specification** (Is all about “Guarantees”)
- **System/Subsystem Design** (Defines API boundaries and behavior)
- **Implementation details** (e.g., “lock-free” and “wait-free” algorithms provide guarantees for system-wide or per-thread progress)
Levels of “Knowing”

Guarantee
- Inviolate principle or behavior

Always true

Examples:
- **C++ Language Specification** (Is all about “Guarantees”)
- **System/Subsystem Design** (Defines API boundaries and behavior)
- **Implementation details** (e.g., “lock-free” and “wait-free” algorithms provide guarantees for system-wide or per-thread progress)

Rule
- Highly regarded principle

Exceptions may apply

Examples:
- **System-specific adapters** may require custom handling
- **Exceptional events** may require special processing
- **Custom or adaptive behavior** may invoke novel execution paths

highest

lowest

The Roles of Symmetry And Orthogonality In Design

Charley Bay - charleyb123 at gmail dot com
Levels of “Knowing”

Guarantee
- Inviolate principle or behavior

Examples:
- **C++ Language Specification** (Is all about “Guarantees”)
- **System/Subsystem Design** (Defines API boundaries and behavior)
- **Implementation details** (e.g., “lock-free” and “wait-free” algorithms provide guarantees for system-wide or per-thread progress)

Rule
- Highly regarded principle

Examples:
- **System-specific adapters** may require custom handling
- **Exceptional events** may require special processing
- **Custom or adaptive behavior** may invoke novel execution paths

Guideline
- General pattern

Examples:
- Prefer generalized solution, but **plugin API** allows for custom processing (such as hardware offloading)
- Prefer default configuration, but **permit users to bypass or disable** specific subsystems
- Customization to adapt system to **customer-specific environment**
Levels of “Knowing”

Guarantee
- Inviolate principle or behavior

Rule
- Highly regarded principle

Guideline
- General pattern

Guess
- Bias projection

Examples:
- **C++ Language Specification** (Is all about “Guarantees”)
- **System/Subsystem Design** (Defines API boundaries and behavior)
- **Implementation details** (e.g., “lock-free” and “wait-free” algorithms provide guarantees for system-wide or per-thread progress)

Examples:
- **System-specific adapters** may require custom handling
- **Exceptional events** may require special processing
- **Custom or adaptive behavior** may invoke novel execution paths

Examples:
- Prefer generalized solution, but **plugin API** allows for custom processing (such as hardware offloading)
- Prefer default configuration, but **permit users to bypass or disable** specific subsystems
- Customization to adapt system to **customer-specific environment**

Projection of personal bias independent of actual system:
- “I don’t know, but this is how I would have done it”
- “Seems like it shouldn’t happen, but it does”
Which “Knowing”? 

- **Given:**

```cpp
... {
  Bar b;
  // ... 
}
... 
```

**Desired:** `b.~Bar()` is called

**Q:** Which “knowing”? 

---

**Guarantee**
- Inviolable principle or behavior

**Rule**
- Highly regarded principle

**Guideline**
- General pattern

**Guess**
- Bias projection

---

**Always true**

**Exceptions may apply**

**Violations not uncommon**

**You don’t know**
Which “Knowing”? 

- **Given:**

```cpp
... 
{  
  Bar b;  
  // ...  
}  
...  
```

**Desired:** `b.~Bar()` is called

**Guaranteed by C++ Language Specification**

**Q:** Which “knowing”? 

- **Guarantee**
  - Inviolate principle or behavior

- **Rule**
  - Highly regarded principle

- **Guideline**
  - General pattern

- **Guess**
  - Bias projection

- **Always true**

- **Exceptions may apply**

- **Violations not uncommon**

- **You don’t know**
Which “Knowing”? 

• Given:

```cpp
...
{
  Bar b;
  // ...
}
...
```

**Desired:**
- `b.~Bar()` is called
- `b` Not in scope

**Guaranteed by C++ Language Specification**

**Q: Which “knowing”?**
Which “Knowing”?  

• Given:

```cpp
...
{
    Bar b;
    // ...
}
...
```

• Desired: `b.Bar()` is called

• Desired: `b` Not in scope

Q: Which “knowing”?  

**Guarantee**
- Inviolate principle or behavior

**Rule**
- Highly regarded principle

**Guideline**
- General pattern

**Guess**
- Bias projection

Always true  

Exceptions may apply  

Violations not uncommon  

You don’t know  

You don’t know
Which “Knowing”?  

- Implement `std::variant<Types...>`

- Desired:  
  - `variant` is “value-type”
  - Implementation cannot allocate dynamic memory

- **BUT!**  
  - discover exception may be thrown during move initialization of contained value (*during move assignment*)

- **SOLUTION:**  
  - `std::variant<Types...>::valueless_by_exception`
Which “Knowing”? 

- Implement `std::variant<Types...>`

Desired:
- `variant` is “value-type”
- Implementation cannot allocate dynamic memory

...BUT!
- discover exception may be thrown during move initialization of contained value (*during move assignment*)

...SOLUTION:
- `std::variant<Types...>::valueless_by_exception`

---

**Highly Regarded Principle:**
`std::variant<Types...>` always has a value
Which “Knowing”?  

- Given:  
  - Concern about `throw-within-uncaught-throw` (`std::terminate()` called)  

- ...SOLUTION:  
  - Never throw within dtor (because stack-unwind during exception handling cannot tolerate a nested throw)  

- Q: Which “knowing”?  

---

**Guarantee**  
- Inviolate principle or behavior  

**Rule**  
- Highly regarded principle  

**Guideline**  
- General pattern  

**Guess**  
- Bias projection  

---

- Always true  
- Exceptions may apply  
- Violations not uncommon  
- You don’t know
Which “Knowing”?

- Given:
  - Concern about `throw-within-uncought-throw` 
    
- ...SOLUTION:
  - Never throw within dtor (because stack-unwind during exception handling cannot tolerate a nested throw)

- Q: Which “knowing”?

If your codebase implements...
- **Rule**: never `throw` in dtor
- **Guideline**: well-defined scenarios may `throw` in dtor
- **Guess**: dtors may `throw`

---

Never a **Guarantee**

...Because no protection against other scenarios (other than dtor) causing `throw-within-uncought-throw`

---

Guarantee
- Inviolate principle or behavior

Rule
- Highly regarded principle

Guideline
- General pattern

Guess
- Bias projection

Always true

Exceptions may apply

Violations not uncommon

You don’t know

cppcon 2021
The Roles of Symmetry And Orthogonality In Design

Charley Bay - charleyb123 at gmail dot com
Role of Symmetry

_Notional understanding without direct inspection_
“Make your point!”
Design Symmetry

Symmetry (def):
Agreement in dimensions due to proportion and arrangement

Symmetric:
- Harmonious or Balanced

Q: Why is Symmetry good (for Design)?
Design Symmetry

Symmetry (def): Agreement in dimensions due to proportion and arrangement

Symmetric:
- Harmonious or Balanced

Q: Why is Symmetry good (for Design)?

A: Symmetry implies high predictability and consistent behavior (once pattern is recognized)

Enables system scaling (in size and complexity)
Symmetric Does NOT Mean “Sameness”

**Symmetric:**
- …is “similar”
- …is NOT “sameness”

- “Celtic knotwork showing p4 symmetry”
- “Kitchen kaleidoscope quilt block”
- “A fractal-like shape that has reflectional symmetry, rotational symmetry and self-similarity”
- “The ceiling of Lotfollah mosque, Isfahan, Iran has 8-fold symmetries.”

**Types of symmetry (in geometry):**
- Reflectional symmetry
- Rotational symmetry
- Translational symmetry
- Helical symmetry
- Scale symmetry
- Glide reflection symmetry
- Rotoreflection symmetry

Humans are GREAT at pattern recognition (identifying that which is “similar”)

Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We **use symmetry** (from what we “see”) **to intuit** that which we do not see.
Symmetry Examples

- Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

Enterolobium cyclocarpum

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.
Symmetry Examples
• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

Enterolobium cyclocarpum
Symmetry Examples

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see

Enterolobium cyclocarpum
Symmetry Examples

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

Enterolobium cyclocarpum
The Phi Scaling Angle In Nature

- Symmetry **naturally occurs** in our system (and in Nature!)

Where there is symmetry, there is **predictability** (actionable understanding)

Spruce tree is fractal scaling based on phi angle

Pelicans in flight with phi angle

https://cosmometry.net/phi-scaling-angle
Symmetry allows us to “know” things that we otherwise should not know

(by enabling projection over that which is not explicitly inspected)
Symmetry Examples *(continued)*

• Q: Guess what is hidden?

We use symmetry *(from what we “see”)* to intuit that which we do not see

This is ONLY effective WHEN the domain is *structured symmetrically*
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see

This is ONLY effective WHEN the domain is structured symmetrically.
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

This is ONLY effective WHEN the domain is structured symmetrically.
Symmetry Examples (continued)

• Q: Guess what is hidden?

We **use symmetry** (from what we “see”) **to intuit** that which we do not see

This is ONLY effective WHEN the domain is **structured symmetrically**
Symmetry Examples (continued)

• Q: Guess what is hidden?

We **use symmetry** *(from what we “see”)* to **intuit** that which we do not see

This is ONLY effective WHEN the domain is **structured symmetrically**
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

This is ONLY effective WHEN the domain is structured symmetrically.
We use symmetry (from what we “see”) to intuit that which we do not see.

This is ONLY effective WHEN the domain is structured symmetrically.
Symmetry Examples (continued)

• Q: Guess what is hidden?

We **use symmetry** (from what we “see”) **to intuit** that which we do not see

This is ONLY effective WHEN the domain is **structured symmetrically**
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

This is ONLY effective WHEN the domain is structured symmetrically.
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry (from what we “see”) to intuit that which we do not see.

This is ONLY effective WHEN the domain is structured symmetrically.
Symmetry Examples (continued)

• Q: Guess what is hidden?

We use symmetry *(from what we “see”)* to intuit that which we do not see.

This is ONLY effective WHEN the domain is *structured symmetrically*. 
Symmetry Examples (continued)

• Q: Guess what is hidden?

We **use symmetry** *(from what we “see”)* **to intuit** that which we do not see.

This is ONLY effective **WHEN** the domain is **structured symmetrically**.
Symmetry In C++ Code

- C++’s most common symmetry example: Resource Management

**Stack-based** *(automatic)* data objects

- Is symmetry to define state based on control-flow *(static lexical scoping)*
- Edge cases managed by the C++ Standard *(Guaranteed!)*

"The compiler giveth, and the compiler taketh away"

```cpp
...
{
    Bar b;
    //...do stuff with b
}
```

Enter the block, object is created

Leave the block, object is destroyed
Symmetry In C++ Code

• C++’s most common symmetry example: Resource Management

**Stack-based** *(automatic)* data objects
- Is symmetry to define state based on control-flow *(static lexical scoping)*
- Edge cases managed by the C++ Standard *(Guaranteed!)*

“The compiler giveth, and the compiler taketh away”

• **Heap-based** *(dynamic)* data objects
- Is symmetry to define state independent of control-flow *(static lexical scoping)*
- Edge cases managed by the developer

“The developer giveth, and the developer better clean up after oneself”

Can implement designs where state escapes compiler-defined control flow governed by the C++ Standard

```cpp
{  
Bar b;  
//...do stuff with b  
}
```

Enter the block, object is created

Leave the block, object is destroyed

```cpp
{  
...  
Bar* b = new Bar();  
ConsumeBar(*b);  
}  
void ConsumeBar(Bar& b) {  
ProcessBar(b);  
delete b;  //...consume  
}
```
Symmetry of the C++ Stack

Stack-based objects have outstanding lifecycle symmetry (Guaranteed!)

```
Bar b0;
//...do stuff with b0
```

Use whenever possible!
Symmetry of the C++ Stack

Stack-based objects have outstanding lifecycle symmetry (Guaranteed!)

Use whenever possible!

```
...{  //...do stuff with b0
    Bar b0;
    //...do stuff with b0
    {
        Bar b1;
        //...do stuff with b0, b1
    }
    //...do stuff with b0, b1

    //...do stuff with b0
}
```
Symmetry of the C++ Stack

Stack-based objects have 
**outstanding lifecycle symmetry**

(Guaranteed!)

```
... {
  Bar b0;
  //...do stuff with b0
  {
    Bar b1;
    //...do stuff with b0, b1
    {
      Bar b2;
      //...do stuff with b0, b1, b2
    }
    //...do stuff with b0, b1
  }
  //...do stuff with b0
}
```

Use whenever possible!
Symmetry of the C++ Stack

Stack-based objects have **outstanding lifecycle symmetry** (Guaranteed!)

Use whenever possible!

Value Semantics is **preferred** in Modern C++, due to superior symmetry in object lifecycle
Symmetry of the C++ Heap

Heap-based objects have lifecycle symmetry independent of (stack-based) control-flow

```cpp
{...{
    Bar* b = new Bar();
    //...do stuff with b
    {
        //...arbitrary code
        DoThing0(*b);
        DoThing1(*b);
    }
    ConsumeBar(*b);
}
```

```cpp
void DoThing0(Bar& b)
{
    //...do stuff with b
}

void DoThing1(Bar& b)
{
    //...do stuff with b
}

void ConsumeBar(Bar& b)
{
    //...do stuff with b
    ProcessBar(b);
    delete b; //...consume
}
```
Symmetry of the C++ Heap

Heap-based objects have lifecycle symmetry independent of (stack-based) control-flow

Your Design defines object lifecycle
• std::unique<> is an implementation tool, not a design decision
The Roles of Symmetry and Orthogonality In Design

Heap-based objects have lifecycle symmetry independent of (stack-based) control-flow.

This is a (useful!) design feature.

Using the heap demands *Design attention* to define and implement object lifecycle.

Your Design defines object lifecycle:
- `std::unique<>` is an implementation tool, *not* a design decision.
The #1 Reason to go to C++ (from C):

(Strong!) Object Lifecycle Symmetry

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
</table>
| **Memory is a “Bucket of bits”**  
*Well-defined*: Type punning, ptr-casting, memory copying | **Memory holds Objects**  
*Well-defined*: (Very!) Strong Object Model (*ctor...dtor*), explicit rules coercing among types within the type system |
Use Design To Establish Symmetry

• We use “Design” to **establish Symmetry**
• *Example*: Restore Symmetry when using `new...delete`

These are **generalized patterns** (variations may apply)

Manager

Factory

Instantiator that dynamically allocates from heap, transferring ownership to the caller

Instantiator that dynamically allocates from heap and which retains ownership *(caller receives handle or light reference)*

Symmetry Restored!

**Manager** type provides options for:

• No leaked instances
• Allocation amortization
• Whole-system reset
• Resource prioritization and recovery
• Resource metrics and runtime monitoring

Composable!
Use Design To Establish Symmetry

- We use “Design” to **establish Symmetry**
- **Example:** Restore Symmetry when using `new...delete`

**Source...Sink:**
- Single location for `new`
- Single location for `delete`
- Well-established resource transfer

**Symmetry Restored!**
- Resource lifecycle is **well-defined**

**Manager** type provides options for:
- No leaked instances
- Allocation amortization
- Whole-system reset
- Resource prioritization and recovery
- Resource metrics and runtime monitoring

**Instantiator** that dynamically allocates from heap, transferring ownership to the caller

**Instantiator** that dynamically allocates from heap and which retains ownership *(caller receives handle or light reference)*

**Composable!**
Use Design To Establish Symmetry

- We use “Design” to **establish Symmetry**
- *Example:* Restore Symmetry when using `new...delete`

**Manager**

- Instantiator that dynamically allocates from heap, transferring ownership to the caller
- Instantiator that dynamically allocates from heap and which retains ownership (caller receives handle or light reference)

**Source...Sink:**

- Single location for `new`
- Single location for `delete`
- Well-established resource transfer

**Symmetry Restored!**

- Resource lifecycle is **well-defined**

**Producer...Consumer:**

- Uses Source...Sink
- Typically, across threads or subsystems

**Symmetry Restored!**

- Manager type provides options for:
  - No leaked instances
  - Allocation amortization
  - Whole-system reset
  - Resource prioritization and recovery
  - Resource metrics and runtime monitoring

**Factory**

- Source performs `new` (and ownership transfer)
- Performs `delete` after instance processing

**Composable!**

These are **generalized patterns** (variations may apply)
Expressing Symmetry Through API

C++ Object Lifecycle
(for class-type) (since 1983)

ctor() Object Lifecycle dtor()

Goal: Find a way to emphasize symmetry in your Design and API

...So Design or API is “obvious”
Expressing Symmetry Through API

C++ Object Lifecycle (for class-type)
- `ctor()`
- `dtor()`

Java Applet Lifecycle (since 1995)
- `init()`
- `destroy()`
- `start()`
- `stop()`
- `paint()`
- `idle()`
- `running()`
- `dead()`

Java Runtime alloc

Goal: Find a way to emphasize symmetry in your Design and API

…So Design or API is “obvious”
Expressing Symmetry Through API

C++ Object Lifecycle
(for class-type)

Object Lifecycle

(Constructor)

(Destructor)

Java Applet Lifecycle
(since 1995)

(Born)

(Idle)

(Running)

(Dead)

(init())

(destroy())

(start())

(stop())

(paint())

Java Runtime alloc

(C++ Object Lifecycle
(since 1983)

(main)

(ctor)

(dtor)

(C++ std::thread
(since 2011)

(thread)

(task[])

(task)

(join)

(join)

Goal: Find a way to emphasize symmetry in your Design and API

...So Design or API is “obvious”
Investigating Symmetry

Example questions to investigate symmetry:

- Does each step express clear purpose?
  - If yes, is more obvious for enforcing step semantics

- Are the steps symmetric?
  - If yes, is more obvious for where a desired action should be placed

- Are the steps guaranteed to occur?
  - If yes, complexity is reduced (edge cases are removed)

- Can a step be “empty” (e.g., “do nothing” or “default” behavior is sufficient)?
  - If yes, becomes easier to use correctly

  If yes, may introduce edge cases and complexity:
  - Can a step be conditionally skipped?
  - Can the steps be reordered?
  - Can new (user-custom) steps be inserted?

Domain-specific (or Application-specific) probing of your components and subsystems will identify changes to lower system complexity

Decreasing Complexity

Increasing Complexity
Role of Asymmetry

*Cheating Symmetry For Fun And Profit*
Cheating Symmetry: `std::move`

**Motivation for `std::move` (since C++11):**
To violate symmetry for gains in efficiency (i.e., state pilfering)

```
Bar bar0;
    //...populate bar0
Bar bar1 = std::move(bar0);
```

Before `std::move`

After `std::move`

“Valid but Unspecified state”

Pilfered state
Motivation for `std::move` (since C++11):
To **violate symmetry** for gains in efficiency (i.e., state pilfering)

- **std::move** is tricky because:
  - Is **NOT symmetric** (pilfered-from object has Unspecified mutation)
  - Is **NOT orthogonal** (pilfered-from and pilfered-into objects are related)

`std::move` is tricky because it represents a **symmetry violation**
Cheating Resource Symmetry

- **C++ Techniques** to cheat object lifecycle symmetry:
  - *(Named-)*Return Value Optimization (**RVO, NRVO**) to transfer instance
  - “Pilfer” or transfer object state:
    - `xvalues` `&&` *(since C++11)*
    - `std::move<>` *(since C++11)*
  - “Light-reference” state managed elsewhere:
    - `std::string_view` *(since C++17)*
    - `std::span` *(since C++20)*

**Tricky:**
- Edge cases are introduced due to **symmetry violations** *(of object lifecycle)*

https://www.youtube.com/watch?v=hA1WNtNyNbo

CppCon 2018: Arthur O’Dwyer
“Return Value Optimization: Harder Than It Looks”
https://www.cppstories.com/2013/02/smart-pointers-gotchas/
The name for a symmetry violation: Asymmetry
Asymmetry

- If a relation exists which is not symmetric, then it is **asymmetric**

Symmetry *(def):* Agreement in dimensions due to proportion and arrangement

**Symmetric**
- Harmonious or Balanced

  *Implies high predictability (when pattern is recognized)*

**Asymmetric**
- Unbalanced or Exceptional

  *Implies edge-cases or surprising behavior “at-scale”*
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry prevents intuiting about what we do not see (based on what we see)
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry preventsintuiting about what we do not see (based on what we see)
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry prevents intuiting about what we do not see (based on what we see)
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry **prevents intuitions** about what we do not see *(based on what we see)*.
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry **prevents intuitions** about what we do not see *(based on what we see)*

https://www.flickr.com/photos/128139955@N02/50811516651
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry **prevents intuitions** about what we do not see *(based on what we see)*

https://www.flickr.com/photos/128139955@N02/50811516651
Asymmetry Examples

- Q: Guess what is hidden?

Asymmetry **prevents intuiting** about what we do not see *(based on what we see)*

https://www.flickr.com/photos/128139955@N02/50811516651
Asymmetry Examples

- Q: Guess what is hidden?

Asymmetry prevents intuitions about what we do not see 
(based on what we see)

https://www.flickr.com/photos/128139955@N02/50811516651
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry prevents intuiting about what we do not see (based on what we see)

https://www.flickr.com/photos/128139955@N02/50811516651
Asymmetry Examples

• Q: Guess what is hidden?

Asymmetry prevents intuiting about what we do not see (based on what we see)
Where is Asymmetry Used?

- Asymmetry may be considered:

1. **To Gain Efficiencies** *(examples):*
   - Short-circuit control flow
   - Resource transfer *(state pilfering)*
   - Object lifecycle extension

   **Take care to not abuse this (it will bite you at-scale)**

2. **To Implement Adapter Layers** *(assembling or adapting among subsystems):*
   - Asymmetric **type-transform** *(mapping 1:N or N:1 data types across API boundaries)*
   - Asymmetric **data serialization** *(mapping 1:N or N:1 data objects for marshalling or serializing across API boundaries)*
   - Asymmetric **control flow adaptation** *(mapping 1:N or N:1 API calls across API boundaries)*
   - Asymmetric **coordination of threads or event models** *(across subsystem boundaries (synchronous or asynchronous))
“

I use asymmetry all the time, and it’s never been a problem.

"
Concerns About Asymmetry

**Desired system attributes (a sampling):**

- **Logical:** People understand it, and is intuitive with minimal onboarding time and effort
- **Implementable:** Complexity is manageable; resource contention is reasonable; and execution model is sufficient for the system to perform as needed
- **Efficient:** Resources are spent wisely
- **Maintainable:** Remains manageable as system is grown or evolved in complexity and size
- **Scalable:** Subsystem linkages continue to robustly perform when under increased stress and load
- **Adaptable:** Remains manageable as system is adapted to new domains
- **Unsurprising:** Predicted behavior is typically the actual behavior, and edge cases are uncommon

**Asymmetry tends to:**

- **violate** desired system aspects
- **prevent** development of system intuition
Asymmetry In C

- Examples of **asymmetry in C Language**: 
  - `struct` (*but not array*) **may be returned** from a function
  - **Array can be returned** if is inside a `struct`
  - `struct` member **can be any data type** (*but not void*)
  - **Default is pass-by-value, except for array** (*which is implicit pass-by-reference*)

These are probably “obvious” to experienced developers 
(*but sometimes surprising to new developers*)
Asymmetry In C++

Asymmetry tends to cause **edge cases** and **surprising behavior**

Examples of **asymmetry in C++**:

- **Short-circuiting** of || and && (all operands may not be evaluated)
- Unspecified evaluation order for **function parameters** (side effects occur in unspecified order)
- **std::shared_ptr<>** (object lifecycle varies depending on handles to instance)
- C++ Standard rules for **object lifecycle extension**
- **Copy elision** (mandatory or non-mandatory elision of copy/move operations)
- **Object Storage Reuse** (std::launder, Undefined to reuse static or const memory)
- Member-function binary operator overloads (left-operand is always *this)
- (Named-)Return Value Optimization (RVO, NRVO) to transfer instance
- xvalues (&&) (since C++11) to pilfer or transfer instance state
- **std::move<>** (since C++11) to transfer instance state
- **std::string_view** (since C++17) for light-reference to external state
- **std::span** (since C++20) for light-reference to external state
- **Coroutines** (since C++20) for control-flow asymmetry such as suspended function (e.g., return, co_return, co_yield, co_await)

---

**std::string_view vs. std::span<>**

- Similar motivations and use cases:
  - Non-owning (*light-reference*) for bounds-safe view to contiguous element sequence
  - span<> is template (string_view is not)
  - string_view is read-only (span<> may modify target elements)
  - string_view supports std::string-like operations (substr, find, compare, ==, <, >)
  - span<> is not Regular, and does not support ==, <, > (see: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1085r2.md)

---

**C++ Globals**

- **Construction order**:
  - Thread-safe for static instance at block scope (since C++11)
  - Reentrancy invokes undefined behavior, see: https://devblogs.microsoft.com/oldnewthing/20040308-00/?p=40363
  - Uncontrolled initialization order for static instance at global scope
- **Destruction order**:
  - Uncontrolled dtor order (may invoke undefined behavior if dependencies are violated across system global state)

---

(Use of) “Some” asymmetries in Your System is probably fine

(Use of) “Lots” of asymmetries is why new developers sometimes fear C++
Asymmetry In Your Codebase

• Asymmetry examples in **Your Codebase**:

  1. **Design Hygiene**
     - **Unclear control-flow**
       - *Example:* Integration across subsystems is unnecessarily complex
     - **Unclear resource management**
       - *Example:* Resource lifecycle is conditional or unnecessarily complex
     - **Weak Abstractions**
       - *Example:* Component API is defined by external demands, not through internal cohesive purpose *(such as: Adapter component)*

  2. **Implementation Hygiene**
     - **Unnecessarily complex processing**
       - *Example:* Eager-compute or Lazy-compute that introduces stochastic time-shifting of computation and resource contention, or which encourages branch mis-prediction
     - **Multiple function returns** *(resulting in different control flows within function)*
     - **Multiple abstraction levels within a given function body**
       “Every statement in a function body should be at the same level of abstraction.” *(paraphrased)*

Abuse of class hierarchies may be “design” or “implementation” *(next page)*
Review: Storage Duration

- **C++ Storage Duration** is one of:
  - **Automatic** *(block-begin…block-end)*
  - **Static** *(program-begin…program-end)*
  - **Dynamic** *(new…delete)*
  - **Thread** *(thread-begin…thread-end)*

Register storage duration is automatic plus compiler hints *(until C++17)*
Review: Storage Duration

- **C++ Storage Duration** is one of:
  - **Automatic** (*block-begin…block-end*)
  - **Static** (*program-begin…program-end*)
  - **Dynamic** (*new…delete*)
  - **Thread** (*thread-begin…thread-end*)

Register storage duration is automatic plus compiler hints (*until C++17*)

```
// ... Bar b;
```

Object Lifetime

- **Lifetime begins** when initialization is complete
- **Lifetime ends** when destructor call starts

```
Bar::Bar() {}
Bar::~Bar() {}
```

Allocating

- constructing

Destructing

- destructing

Deallocating
Review: Storage Duration

**C++ Storage Duration** is one of:

- **Automatic** *(block-begin…block-end)*
- **Static** *(program-begin…program-end)*
- **Dynamic** *(new…delete)*
- **Thread** *(thread-begin…thread-end)*

Register storage duration is automatic plus compiler hints *(until C++17)*

Surprising behavior (“bugs”) may occur when accessing an object outside its lifetime.

```cpp
// ... Bar b;

Bar::Bar() {}
Bar::~Bar() {}
```

**Object Lifetime** *(object can be used without limitations)*

- **Lifetime begins** when initialization is complete
- **Lifetime ends** when destructor call starts

**Life Cycle**

- **Allocating**
- **Constructing**
- **Object Lifetime**
- **Destructing**
- **Deallocating**
Review: Storage Duration

C++ Storage Duration is one of:

- **Automatic** *(block-begin…block-end)*
- **Static** *(program-begin…program-end)*
- **Dynamic** *(new…delete)*
- **Thread** *(thread-begin…thread-end)*

Object Lifetime Begins when BOTH of:
1. Storage is obtained
2. Initialization is complete

Object Lifetime Ends when EITHER of:
1. Dtor starts
2. Storage is reused or released

Register: storage duration is automatic plus compiler hints *(until C++17)*

Surprising behavior (“bugs”) may occur when accessing an object outside its lifetime.
Review: Storage Duration

- **C++ Storage Duration** is one of:
  - **Automatic** *(block-begin…block-end)*
  - **Static** *(program-begin…program-end)*
  - **Dynamic** *(new…delete)*
  - **Thread** *(thread-begin…thread-end)*

---

**Object Lifetime**

**Begins** when BOTH of:
1. Storage is obtained
2. Initialization is complete

**Ends** when EITHER of:
1. Dtor starts
2. Storage is reused or released

Register storage duration is automatic plus compiler hints *(until C++17)*

---

Surprising behavior *(“bugs”)* may occur when accessing an object outside its lifetime

---

Watch out for **asymmetries** from:
- **std::move()** *(i.e., leaving “valid but unspecified state”)*
- **Temporary objects** *(i.e., prvalue “materialization”)*
- **xvalues** *(“eXpiring values”)*
- **RVO, NRVO** *(Named-Return Value Optimization)*
Hierarchy Hygiene

C++ inheritance exhibits outstanding symmetry (Guaranteed!)

```cpp
class A {
    public:
    virtual void f();
    int a;
};

class B : public A {
    public:
    void f() override;
    int b;
};

class C : public B {
    public:
    void f() override;
    int c;
};
```
Hierarchy Hygiene

**C++ inheritance exhibits outstanding symmetry** *(Guaranteed!)*

Special **edge cases** exist for hierarchy **implementation**

Why? Ctor and dtor execute **outside** of the **Object Lifetime**

```cpp
class A {
    public:
        virtual void f();
        int a;
};
class B : public A {
    public:
        void f() override;
        int b;
};
class C : public B {
    public:
        void f() override;
        int c;
};
```
“During construction or destruction, the more derived classes do not exist.”

Leads to **General Practice** (pick one):

1. “Never call virtual functions during construction or destruction”

2. “Virtual functions aren’t virtual during construction and destruction”
Hierarchy Hygiene

- **C++ inheritance** exhibits outstanding symmetry *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

```
// ... C my_c;
```

```
class A {
    public:
        virtual void f();
        int a;
};
class B : public A {
    public:
        void f() override;
        int b;
};
class C : public B {
    public:
        void f() override;
        int c;
};
```

```
A::A() : a(1) { f(); } //A::f()
```

```
memory
my_c
```

```
A
B
C
```

Symmetrical in C++, but Your Implementation may have assumed **asymmetrical**
Hierarchy Hygiene

- **C++ inheritance** exhibits outstanding symmetry (Guaranteed!)

- Special **edge cases** exist for hierarchy implementation

```cpp
// ... 
C my_c;
```

```
class A {
    public:
        virtual void f();
        int a;
};
class B : public A {
    public:
        void f() override;
        int b;
};
class C : public B {
    public:
        void f() override;
        int c;
};
```

```
memory
  my_c

A
B
C
```

Symmetrical in C++, but Your Implementation may have assumed asymmetrical

```cpp
//A::f()
A::A() : a(1) { f(); } //A::f()
B::B() : b(2) { f(); } //B::f()
```

```
A exists
B exists
```
Hierarchy Hygiene

- **C++ inheritance** exhibits **outstanding symmetry** (Guaranteed!)
- **Special edge cases** exist for hierarchy **implementation**

```cpp
class A {
    public:
        virtual void f();
        int a;
};
class B : public A {
    public:
        void f() override;
        int b;
};
class C : public B {
    public:
        void f() override;
        int c;
};
```

```
// ... C my_c;
A::A() : a(1) { f(); } // A::f()
B::B() : b(2) { f(); } // B::f()
C::C() : c(3) { f(); } // C::f()
```

**Object Lifetime**

- **A exists**
- **B exists**
- **C exists**

**Lifetime begins when initialization is complete**

```
// constructing
memory
A
B
C

// allocating
```

**Symmetrical in C++, but Your Implementation may have assumed asymmetrical**
Hierarchy Hygiene

- **C++ inheritance** exhibits outstanding symmetry (Guaranteed!)
- Special **edge cases** exist for hierarchy implementation

```cpp
class A {
    public:
        virtual void f();
        int a;
};
class B : public A {
    public:
        void f() override;
        int b;
};
class C : public B {
    public:
        void f() override;
        int c;
};
```

- **Symmetrical in C++, but** Your Implementation may have assumed **asymmetrical**

```cpp
// ... C my_c;
```

Object Lifetime (object can be used without limitations)

```cpp
A::A() : a(1) { f(); } //A::f()
B::B() : b(2) { f(); } //B::f()
C::C() : c(3) { f(); } //C::f()
```

- **Lifetime begins when** initialization is complete
- **Lifetime ends when** destructor call starts

```cpp
C::~C() {}
```

```cpp
A exists
B exists
C exists
```

```
memory
```

```
my_c
```

```
a b c
```
Hierarchy Hygiene

- **C++ inheritance** exhibits **outstanding symmetry** (Guaranteed!)

- Special **edge cases** exist for hierarchy **implementation**

```cpp
// ... 
C my_c;

class A {
   public:
      virtual void f();
      int a;
};

class B : public A {
   public:
      void f() override;
      int b;
};

class C : public B {
   public:
      void f() override;
      int c;
};
```

```cpp
A exists
B exists
C exists
```

```cpp
//A::f()
A::A() : a(1) { f(); } //A::f()

B::B() : b(2) { f(); } //B::f()

C::C() : c(3) { f(); } //C::f()

//B::f()
B::~B() {}

//C::f()
C::~C() {}
```

```
// Lifetime begins when initialization is complete
// Lifetime ends when destructor call starts
```

```
Symmetrical in C++, but Your Implementation may have assumed asymmetrical
```

```cpp
//Allocating
A::A()
B::B()
C::C()

//Constructing
my_c
```

```cpp
//Deallocating
C my_c;
```

```cpp
//Destructing
B::~B()
C::~C()
```
### Hierarchy Hygiene

- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*

- Special special **edge cases** exist for hierarchy **implementation**

```cpp
// ... C my_c;

A::A() :a(1) { f(); } //A::f()
B::B() :b(2) { f(); } //B::f()
C::C() :c(3) { f(); } //C::f()
```

```
class A {
    public:
        virtual void f();
        int a;
};

class B : public A {
    public:
        void f() override;
        int b;
};

class C : public B {
    public:
        void f() override;
        int c;
};
```

```
//A::f()
A::A() :a(1) { f(); }
B::B() :b(2) { f(); }
C::C() :c(3) { f(); }
```

```
A::~A() {}
B::~B() {}
C::~C() {}
```

**Symmetrical in C++, but Your Implementation may have assumed asymmetrical**

**Object Lifetime** *(object can be used without limitations)*

```
//my_c
A my_c;
```

**Lifetime begins when initialization is complete**

**Lifetime ends when destructor call starts**
Other Asymmetry in Class Hierarchies

- Other examples of class hierarchy implementation asymmetry:
  - **Override asymmetry**: If `Base::~Base()` is not `virtual`, then `delete ptr_to_Base` will not invoke `Derived::~Derived()`
  - **API asymmetry**: Overloads in the `Derived` will “shadow/hide” `virtual` in the `Base` (unless using `Base::name`)
  - **API asymmetry**: Overloads with the same name as overrides can “shadow/hide” the override signature
  - *this used in base/member initializer list*:
    - Take care to not access `Derived` members in the `Base::Base()` (because derived ctor did not start, so derived members do not exist, so is `Undefined` behavior to access members)
    - **Why? Because**: What if `virtual` inheritance is used where `Base::Base` needed the `vptr` to access the `Derived` member?

With attention *(or practice)*, it becomes easy to identify asymmetry within our implementation.
Defining C++ Concepts (Since C++11)

Q: What is the role of Symmetry in defining a concept?

“Good concepts express more than just what an algorithm needs”
-- Jeff Garland

(Example):
• IF: Your concept requires “plus”
• SHOULD: You also requires “minus”?
• BECAUSE:
  • “Addable ➔ likely a poor concept, try Number”
    – Jeff Garland
  • Perhaps want to “plus” a negative number?
  • Perhaps want implementation flexibility (enabling future maintenance)?
  • Math symmetry makes your types consistent, flexible, and adaptable to custom algorithms (and is implied for optimization through compiler canonicalization)
Role of Orthogonality

Removing edge cases and coupling by making things unrelated
"I’m Old School,
Are you sure this is useful?"
Orthogonality In Programming

- Concept introduced to programming in the design of Algol 68 (1968)
- Guarantees that modifying a component does not create nor propagate side effects to other components
- Essential for design of complex systems:
  - System becomes implementable
  - Emergent system behavior is strictly controlled by logic (not by side effects of integration artifacts)
- Reduces testing and development time (because is easier to verify designs that do not cause nor depend upon side effects)
- Achieved through:
  - Separation of Concerns
  - Encapsulation

Want to Learn More? See: Edsger W. Dijkstra Separation of Concerns

The number of independent primitive concepts has been minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied “orthogonally” in order to maximize the expressive power of the language while trying to avoid deleterious superfluities.

---

Adriaan "Aad" van Wijngaarden (1916-1987)

Dutch mathematician and computer scientist:
- Numerical analysis
- Programming languages
- Design principles
Orthogonality In Practice

Orthogonal:
- Unrelated *(no relation exists)*

Orthogonal goal:
- Have composable units **without surprising cross-linkages**

Orthogonal components:
- Can be **used independent of context**
- Can be used in **arbitrary combinations** with **consistent results**

Orthogonal design:
- Associated with **simplicity** *(the more orthogonal the design, the fewer the exceptions)*

Orthogonality grants simplicity to **dismiss as a possibility** some behaviors or component interactions within the resulting system.

**We use orthogonality to:**
1. **Remove interactions**
2. **Reduce coupling**

**Benefits:**
- Less complexity
- Fewer edge cases
- Increased stability
- Greater reuse
- Better scaling

Orthogonal *(def)*:
1. *(mathematics)* Perpendicular
2. *(programming)* Unrelated

Today, orthogonality is used in:
- Design of instruction sets
- Design of programming languages
- Design of APIs
- Design of user interfaces

At right-angles

Make things “unrelated”
Orthogonality allows us to dismiss as a possibility some behaviors or component interactions

(without tedious inspection)
Case Study: Thread-Stealing Work Queue

Thread-Stealing Work Queue
A design (or pattern) for atomizing or distributing work
• Classic Pattern (very old, and re-discovered many times)
• Is Well-Understood (common understanding for How It Works™, and possible variations)
• Is Highly Robust (correctly, properly, and robustly applied because we know what it solves and how to defend against edge cases)

Is great design because is both symmetrical and orthogonal
Thread-Stealing Work Queue

A design (or pattern) for atomizing or distributing work

- **Classic Pattern** (very old, and re-discovered many times)
- **Is Well-Understood** (common understanding for How It Works™, and possible variations)
- **Is Highly Robust** (correctly, properly, and robustly applied because we know what it solves and how to defend against edge cases)

Is great design because is both symmetrical and orthogonal

**Symmetry:**

**Producer ➔ Consumer:** Work items are arbitrarily produced; and each is consumed exactly once

- **Benefits:** Creating work is obviously correlated with understanding for how work is completed (consumed)
- **Costs:** Special handling is required to violate design symmetry when asymmetry is desired (such as special handling to execute one work item many times)
**Thread-Stealing Work Queue**

A design (or pattern) for **atomizing** or **distributing work**

- **Classic Pattern** (very old, and re-discovered many times)
- **Is Well-Understood** (common understanding for How It Works™, and possible variations)
- **Is Highly Robust** (correctly, properly, and robustly applied because we know what it solves and how to defend against edge cases)

Is great design because is both symmetrical and orthogonal

**Symmetry:**
- **Producer ➔ Consumer:** Work items are arbitrarily produced; and each is consumed exactly once
  - **Benefits:** Creating work is obviously correlated with understanding for how work is completed (consumed)
  - **Costs:** Special handling is required to violate design symmetry when asymmetry is desired (such as special handling to execute one work item many times)

**Orthogonality:**
- **Work Item execution (consumption) is unrelated** to the producer (e.g., execution is delegated to “work-engine” composed of queues and threads)
  - **Benefits:** Greater scaling as threads/workers/resources are made available
  - **Costs:** Producer cannot directly monitor work progress (but indirect monitoring can be implemented)
Orthogonal Design Examples

- Designs leveraging **orthogonal behavior**:

  **Thread-stealing Work Queue**: RAII work items transferred to queue, which are consumed by worker threads

  **Orthogonal Because**: Work item creation is “orthogonal” (unrelated) to work item processing

  **Independent Agent**: RAII work items are instantiated as independent actors, which autonomously progress through a lifecycle or are early-terminated

  **Orthogonal Because**: Work item creation is “orthogonal” (unrelated) to work item processing

  **Trigger-interface APIs**: Trigger, handler, or callback invokes subsystem orthogonally to normal system execution flow (e.g., handling raised exceptions, system events, queued callbacks)

  **Orthogonal Because**: API is invoked orthogonal to normal system control flow

  **Resource Sharing APIs**: Multiple subsystems share access to the same resource (implementation detail may rely upon external manager, or std::shared_ptr to ensure lifecycle of shared resource)

  **Orthogonal Because**: Resource lifecycle is orthogonal to resource usage over time
Orthogonal Design Examples (continued)

- Designs leveraging **orthogonal behavior**:

  - **Implementation Bridge**: API provides no interface for an essential internal operation (which is internally bridged through the public interface)
    - Orthogonal Because: Public interface is independent of internal execution and implementation
    - Example: Synchronous/Asynchronous bridge (such as to implement proactive or reactive read/write)

  - **Fire-And-Forget**: Function immediate-return with work transferred to alternate thread
    - Orthogonal Because: Function call is orthogonal to execution time required to perform the operation implementation
    - Example: High-Speed Logging (immediate-return, where implementation is transferred to a thread other than that which made the call)

  - **Execution Interface**: Control flow proceeds through composition of custom types adhering to defined interface
    - Orthogonal Because: Application-specific control flow and custom types are defined orthogonal to system invocation
    - Examples: Plugin APIs, Dependency Injection of subsystems
Design Relationships

Orthogonal, Symmetric, or Asymmetric
A **Design Relationship** is the degree to which a component relies upon another component ("is aware of")
Can you map that out for me?
Design Relationships

1 Orthogonal
• No relation exists

**PREFERRED** whenever possible
• Lower component coupling
• Greater flexibility *(more implementation options)*
• Higher component reuse
• More robust system under load
• Greater system scaling
Design Relationships

1. **Orthogonal**
   - No relation exists
   - Lower component coupling
   - Greater flexibility (*more implementation options*)
   - Higher component reuse
   - More robust system under load
   - Greater system scaling

2. **Symmetric**
   - Relation is balanced or harmonious
   - Consistent behavior
   - Can intuit what we do not see (*from what we do see*)
   - System scales in size and complexity

**PREFERED** whenever possible

**DESIRER**
Design Relationships

1. **Orthogonal**
   - No relation exists
   - Preferred whenever possible
   - Lower component coupling
   - Greater flexibility (*more implementation options*)
   - Higher component reuse
   - More robust system under load
   - Greater system scaling

2. **Symmetric**
   - Relation is balanced or harmonious
   - Desired
   - Consistent behavior
   - Can intuit what we do not see (*from what we do see*)
   - System scales in size and complexity

3. **Asymmetric**
   - Relation is unbalanced or exceptional
   - Discouraged
   - Has edge cases, surprising behavior
   - Cannot intuit what we do not see
   - Difficult to scale system in size and complexity
Design Relationships

1. **Orthogonal**
   - No relation exists
   - PREFERED whenever possible
   - Lower component coupling
   - Greater flexibility *(more implementation options)*
   - Higher component reuse
   - More robust system under load
   - Greater system scaling

2. **Symmetric**
   - Relation is balanced or harmonious
   - DESIRED
   - Consistent behavior
   - Can intuit what we do not see *(from what we do see)*
   - System scales in size and complexity

3. **Asymmetric**
   - Relation is unbalanced or exceptional
   - DISCOURAGED
   - Has edge cases, surprising behavior
   - Cannot intuit what we do not see
   - Difficult to scale system in size and complexity
Relationship Strength

Given $A$, I know $B$ (Strong Relationship is guaranteed)

- Example: $B$ is computed from $A$

Useful and Robust!
Relationship Strength

Given $\mathbf{A}$, I know $\mathbf{B}$

**Strong Relationship**

(is guaranteed)

- Example: $\mathbf{B}$ is computed from $\mathbf{A}$

**Useful and Robust!**

---

Given $\mathbf{A}$, I know *nothing* about $\mathbf{B}$

- Example: $\mathbf{B}$ and $\mathbf{A}$ exist in different threads within thread-local storage

**No Relationship**

($\mathbf{B}$ is orthogonal to $\mathbf{A}$)

**Useful and Robust!**
Relationship Strength

Given **A**, I know **B** **Strong Relationship** (is guaranteed)
- Example: **B** is computed from **A**

Given **A**, I know *something* about **B** **Weak Relationship** (relationship not guaranteed)
- Example: **B** tends express based on value and amplitude of **A** (or to **A** deltas)
  - **B** value may have *causation* or *correlation* to one or both of:
    - **A** value, **A** deltas (such as increments)

Given **A**, I know *nothing* about **B** **No Relationship** (**B** is orthogonal to **A**)
- Example: **B** and **A** exist in different threads within thread-local storage

Useful and Robust!
Relationship Strength

Given **A**, I know **B**

**Strong Relationship** *(is guaranteed)*
- Example: **B** is computed from **A**

**Useful and Robust!**

Given **A**, I know **something** about **B**

**Weak Relationship** *(relationship not guaranteed)*
- Example: **B** tends express based on value and amplitude of **A** *(or to **A** deltas)*
- **B** value may have causation or correlation to one or both of:
  - **A** value, **A** deltas (such as increments)
- Correlations imply dependencies *(with implications for scalability and side-effects)*
- Assumptions may be invalid for your scenario

**Tricky!**

Given **A**, I know **nothing** about **B**

**No Relationship** *(**B** is orthogonal to **A)***
- Example: **B** and **A** exist in different threads within thread-local storage

**Useful and Robust!**

• Correlations imply dependencies *(with implications for scalability and side-effects)*
• Assumptions may be invalid for your scenario
Relationship Strength

Given **A**, I know **B**

**Strong Relationship** (is guaranteed)
- Example: **B** is computed from **A**

Useful and Robust!

Given **A**, I know *something* about **B**

**Weak Relationship** (relationship not guaranteed)
- Example: **B** tends to express based on value and amplitude of **A** (or to **A** deltas)

- **B** value may have *causation* or *correlation* to one or both of:
  - **A** value, **A** deltas (such as increments)

- Correlations *imply dependencies* (with implications for scalability and side-effects)
- Assumptions may be invalid for your scenario

Tricky!

Given **A**, I know *nothing* about **B**

**No Relationship** (**B** is orthogonal to **A**)
- Example: **B** and **A** exist in different threads within thread-local storage

Useful and Robust!

Knowing “something” can be more dangerous than knowing “nothing” (orthogonality provides stronger guarantees)
Prefer Stronger Two-Way Relationships

- **Prefer Stronger Guarantees** *(one of):*
  - *(strong-)*Symmetry
  - Orthogonality

**Why?**
- Greater “Knowing”
- Fewer assumptions
- Reduced edge cases

Given **A**, I know **B**
Given **B**, I know **A**

Given **A**, I know **nothing** about **B**
Given **B**, I know **nothing** about **A**

Design work may be required to make a relationship orthogonal *(to gain stronger guarantees)*
Prefer Stronger Two-Way Relationships

- Prefer Stronger Guarantees (one of):
  - (strong-)Symmetry
  - Orthogonality

Why?
- Greater “Knowing”
- Fewer assumptions
- Reduced edge cases

Design work may be required to make a relationship orthogonal (to gain stronger guarantees)

Bidirectional Symmetry
Relationships

*In Theory…*

<table>
<thead>
<tr>
<th>Relationship Exists?</th>
<th>Symmetric</th>
<th>Asymmetric</th>
<th>Orthogonal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Relationships can be *complicated*.
The Roles of Symmetry And Orthogonality In Design

In Theory…

Relationship Exists?

Yes
Asymmetric

No
Orthogonal

In Reality…

Relationship Exists?

Yes
Symmetric
Asymmetric

No
Orthogonal

Relationships can be complicated
Relationships

In Theory…

Relationship Exists?

<table>
<thead>
<tr>
<th>Symmetric</th>
<th>Asymmetric</th>
<th>Orthogonal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

In Reality…

Relationship Exists?

<table>
<thead>
<tr>
<th>Symmetric</th>
<th>Asymmetric</th>
<th>Orthogonal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Not perfectly Symmetric

- Create…destroy (e.g., globals)
- Some operations do not fully go backwards
  - Init…uninit
  - Load…unload
  - Start…snapshot-save…shutdown
- Lifecycle Exception or Error Tracing
  - Source…sink
  - Producer…consumer
  - Push…pull (how to leak back failed operation from previous push)

Relationships can be complicated
Relationships

In Theory…

Relationship Exists?

Symmetric
Yes
Asymmetric
No
Orthogonal

In Reality…

Relationship Exists?

Symmetric
Yes
Asymmetric
No
Orthogonal

Not perfectly Symmetric

• Create…destroy (e.g., globals)
• Some operations do not fully go backwards
  • Init…uninit
  • Load…unload
  • Start…snapshot-save…shutdown
• Lifecycle Exception or Error Tracing
  • Source…sink
  • Producer…consumer
  • Push…pull (how to leak back failed operation from previous push)
• Asymmetric patterns exhibiting symmetric correlations
  • Ranged object lifetimes (e.g., `std::smart_ptr<>`, `std::unique_ptr<>`)
  • Time-shifted computation (i.e., lazy-compute, eager-compute)
  • Double-compute (e.g., in iterators, or when using `std::range`)
  • Synchronization of async calls or across threads

Relationships can be complicated
Relationships

In Theory…

Relationship Exists?

<table>
<thead>
<tr>
<th>Symmetric</th>
<th>Asymmetric</th>
<th>Orthogonal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

In Reality…

Relationship Exists?

<table>
<thead>
<tr>
<th>Symmetric</th>
<th>Asymmetric</th>
<th>Orthogonal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Not perfectly Symmetric

- Create...destroy (e.g., globals)
- Some operations do not fully go backwards
  - Init...uninit
  - Load...unload
  - Start...snapshot-save...shutdown
- Lifecycle Exception or Error Tracing
  - Source...sink
  - Producer...consumer
  - Push...pull (how to leak back failed operation from previous push)

Asymmetric patterns exhibiting symmetric correlations

- Ranged object lifetimes (e.g., `std::smart_ptr<>`, `std::unique_ptr<>`)
- Time-shifted computation (i.e., lazy-compute, eager-compute)
- Double-compute (e.g., in iterators, or when using `std::range`)
- Synchronization of async calls or across threads

Orthogonal components exhibiting symmetric correlations

- Custom control flows, data flows
- Special Use Case handling
- Competition for Resources
  - CPU cache pressures
  - Cache line false sharing
  - System calls
  - File handles
  - Network traffic
  - Other OS resources
  - Other hardware resources

Relationships can be complicated
Relationships

In Theory…

<table>
<thead>
<tr>
<th>Relationship Exists?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Orthogonal</td>
</tr>
</tbody>
</table>

In Reality…

<table>
<thead>
<tr>
<th>Relationship Exists?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Orthogonal</td>
</tr>
</tbody>
</table>

Not perfectly Symmetric

- Create…destroy (e.g., globals)
- Some operations do not fully go backwards
  - Init…uninit
  - Load…unload
  - Start…snapshot-save…shutdown
- Lifecycle Exception or Error Tracing
  - Source…sink
  - Producer…consumer
  - Push…pull (how to leak back failed operation from previous push)

Asymmetric patterns exhibiting symmetric correlations

- **Ranged object lifetimes** (e.g., `std::smart_ptr<>`, `std::unique_ptr<>`)
- **Time-shifted computation** (i.e., lazy-compute, eager-compute)
- **Double-compute** (e.g., in iterators, or when using `std::range`)
- **Synchronization** of async calls or across threads

Orthogonal components exhibiting symmetric correlations

- Custom control flows, data flows
- Special Use Case handling
- Competition for Resources
  - CPU cache pressures
  - Cache line false sharing
  - System calls
  - File handles
  - Network traffic
  - Other OS resources
  - Other hardware resources

Orthogonal components exhibiting asymmetric correlations

- System event handling
- Event loop competition (e.g., GUI, network, work-queues, etc.)
- Control flow exceptions

Relationships can be **complicated**
The Roles of Symmetry And Orthogonality In Design

Relationship Attributes:
- **Coupling** (none…indirect…direct)
- **Symmetry** (none…weak…strong)

(Huge!) Design Space

to define relationships

(one-way, two-way)

**Relationship Space**

(orthogonal)   (weak symmetry)   (strong symmetry)

None       Weak           Strong

Relationship Symmetry

Relationship Coupling
Relationship Attributes:
- **Coupling** (none…indirect…direct)
- **Symmetry** (none…weak…strong)

(Huge!) Design Space

to define relationships
(one-way, two-way)

Relationship Space

(Orthogonal)  (Weak Symmetry)  (Strong Symmetry)

None  Weak  Strong
Conclusion

Leverage Symmetry and Orthogonality
Symmetry vs. Orthogonality

**Symmetric**
Means **Similar**
*NOT “the same”*

**Orthogonal**
Means **Unrelated**
*no relationship exists*

These are **NOT** opposites!
Roles In Design

- In Design...

### Role of Symmetry: To make similar (through balance and proportion)
- **Why:** To increase consistency and predictability

### Role of Orthogonality: To make unrelated (non-interacting)
- **Why:** To eliminate possible interactions

IF is NOT symmetric and NOT orthogonal, THEN you have an **Asymmetry** (special pattern or interaction)
- Typically **manifests as edge cases**
- Can be “surprising” at-scale or under system-load
- Can manifest **complex behavior**
  - **Good:** Efficiencies (e.g., `std::move`)
  - **Bad:** Gotchas (e.g., “valid but unspecified”)

YES: Intuitive

NO: Tedious inspection required
Leveraging Symmetry And Orthogonality

• Component Relationships:
  1. **Symmetry** leverages “similarity”
  2. **Orthogonality** leverages “unrelatedness”
  3. In combined consideration, symmetry and orthogonality define **all possible design relationships**

**Relationships can be complicated**

Therefore, whenever possible leverage symmetry and orthogonality as tools to simplify system coupling and dependencies

**Benefits:**
- Less complexity
- Fewer edge cases
- Increased stability
- Greater reuse
- Better scaling
- Enhanced system intuition
Example Relationships

**Example Symmetry (for Data Flows):**
- Flow One-Way: “All data flows left-to-right”
- Flow Wave: “Flow left during computation; flow right during draw-frame”
- Flow Circular: “All components hand-to-the-left (completing a circuit)”

**Example Asymmetry:**
- Thresholding: Processing sometimes short-circuits *(such as when system is under load)*
- Preemption: Work item may be aborted *(perhaps revisited)* if not completed within time limit
- Conditional Reuse: Work item may be sometimes re-used *(if repeat-processing is needed)*

**Example Orthogonality:**
- Entirely independent: Network traffic flows and frame-draw
- Entirely independent: Log traffic processing and main thread
- Entirely independent: Work item processing and allocator amortization
Reducing Risk

Leverage Symmetry and Orthogonality to improve system safety (and reduce risk as presented through surprising interactions)

Inter-Component Relationship Risk

Risk (edge case discovery)
- Danger (more edge cases)
- Safer (fewer edge cases)

Relationship Symmetry
- None
- Weak
- Strong

Inter-component edge cases tend to present when system is:
- At-Scale
- Under-Load

Less edge cases for:
- Symmetric relationships
- Orthogonal relationships

More edge cases for:
- Asymmetric relationships
**Common Design Error:** Establishing unbalanced relationship where:

1. Benefits *(such as efficiency)* do not justify added complexity
2. Unbalanced relationship was accidental *(missed opportunity for one of)*:
   - **Separation of Concerns:** Could have established Orthogonality
   - **Design Elegance:** Could have established Symmetry

---

**Your Design Relationship**

is always one-or-both of:

1. **State** relationship
2. **Control-flow** relationship
Design In-Practice

**Common Design Error:**

**Establishing unbalanced relationship** where:

1. **Benefits (such as efficiency)** do not justify added complexity
2. Unbalanced relationship was accidental *(missed opportunity for one of):*
   - **Separation of Concerns:** Could have established Orthogonality
   - **Design Elegance:** Could have established Symmetry

**Limitations and Surprises:**

Even with balanced relationships, we sometimes see:

- **For Symmetry:** Surprising variations, or edge cases
- **For Orthogonality:** Surprising interactions

---

Your Design Relationship is always one-or-both of:

1. **State** relationship
2. **Control-flow** relationship
Closing Thought:

(For our systems), Symmetry is a processing amplification which is desirable because subsystems in sympathetic resonance manifest complex behavior and computation that is otherwise not achievable.
Thank you! for listening

Questions?

I HAVE QUESTIONS

LOTS OF QUESTIONS