The Roles of Symmetry And Orthogonality In Design

CHARLEY BAY

Cppcon
The C++ Conference
2021
October 24-29

The Roles of
Symmetry And Orthogonality
In Design

charley bay
charleyb123 at gmail dot com

"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_committee_dont_want_to_fix_rangebased_for_loop/heug4br/

Today's Agenda

- Levels of "**Knowing**"
- Role of **Symmetry**
- Role of **Asymmetry**
- Role of **Orthogonality**
- Design **Relationships**
- **Conclusion**

What's going on here?

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?

*Is "**obvious**" or "**clear**"*

- A: We "**Know**":

*Our "**First**" or "**Best Guess**" to any question is **usually correct***

How the structure and behavior achieves a desired result

*We **understand** the inner-workings of our system*

Levels of "Knowing"

highest

Guarantee

Always `true`

- Inviolate principle or behavior

lowest

- *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"

highest

Guarantee

Always `true`

- 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

Exceptions may apply

- 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

lowest

Levels of "Knowing"

highest

Guarantee — *Always true*

- Inviolate principle or behavior

Rule — *Exceptions may apply*

- Highly regarded principle

Guideline — *Violations not uncommon*

- General pattern

lowest

- *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**

Levels of "Knowing"

highest

Guarantee

Always true

- 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

Exceptions may apply

- 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

Violations not uncommon

- 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**

Guess

You don't know

- Bias projection

- 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"*

lowest

Which “Knowing”?

- Given:

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

...
```

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

Guarantee *Always* ***true***
- Inviolate principle or behavior

Rule *Exceptions may apply*
- Highly regarded principle

Guideline *Violations not uncommon*
- General pattern

Guess *You don’t know*
- Bias projection

- Q: Which “knowing”?

Which "Knowing"?

- Given:

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

...
```

Desired:
b.~Bar()
is called

Guaranteed by
C++ Language Specification

Guarantee — *Always* ***true***
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don't know*
- Bias projection

- Q: Which "knowing"?

Which “Knowing”?

- Given:

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

...
```

Desired:
b.~Bar()
is called

Desired:
b
Not in scope

Guaranteed by C++ Language Specification

Guarantee — *Always* ***true***
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don’t know*
- Bias projection

- Q: Which “knowing”?

Which "Knowing"?

- Given:

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

  ...
```

Desired:
b.~Bar()
is called

Desired:
b
Not in scope

Guaranteed by C++ Language Specification

Guaranteed by C++ Language Specification

Guarantee — *Always* ***true***
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don't know*
- Bias projection

- Q: Which "knowing"?

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`

Guarantee *Always* `true`
- Inviolate principle or behavior

Rule *Exceptions may apply*
- Highly regarded principle

Guideline *Violations not uncommon*
- General pattern

Guess *You don’t know*
- Bias projection

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`

Rule through *Desired Semantics* meeting *Implementation Reality*

Guarantee — *Always* `true`
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don't know*
- Bias projection

Highly Regarded Principle:
`std::variant<Types…>` always has a value

Exception: `valueless_by_exception`

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 — *Always `true`*
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don’t know*
- Bias projection

Which "Knowing"?

- Given:
  - Concern about **`throw`**-within-uncaught-**`throw`** *(**`std::terminate()`** called)*
- *…SOLUTION:*
  - <u>**Never throw within dtor**</u> *(because stack-unwind during exception handling cannot tolerate a nested **`throw`**)*
- Q: Which "knowing"?

Guarantee — *Always **`true`***
- Inviolate principle or behavior

Rule — *Exceptions may apply*
- Highly regarded principle

Guideline — *Violations not uncommon*
- General pattern

Guess — *You don't know*
- Bias projection

Never a <u>**Guarantee**</u>
…Because no protection against other scenarios *(other than dtor)* causing **`throw`**-within-uncaught-**`throw`**

If your codebase implements…
- <u>**Rule**</u>: never **`throw`** in dtor
- <u>**Guideline**</u>: well-defined scenarios may **`throw`** in dtor
- <u>**Guess**</u>: dtors may **`throw`**

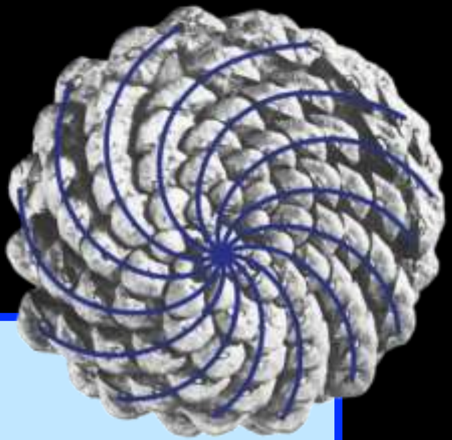
Role of Symmetry

Notional understanding without direct inspection

> *Make your point!*

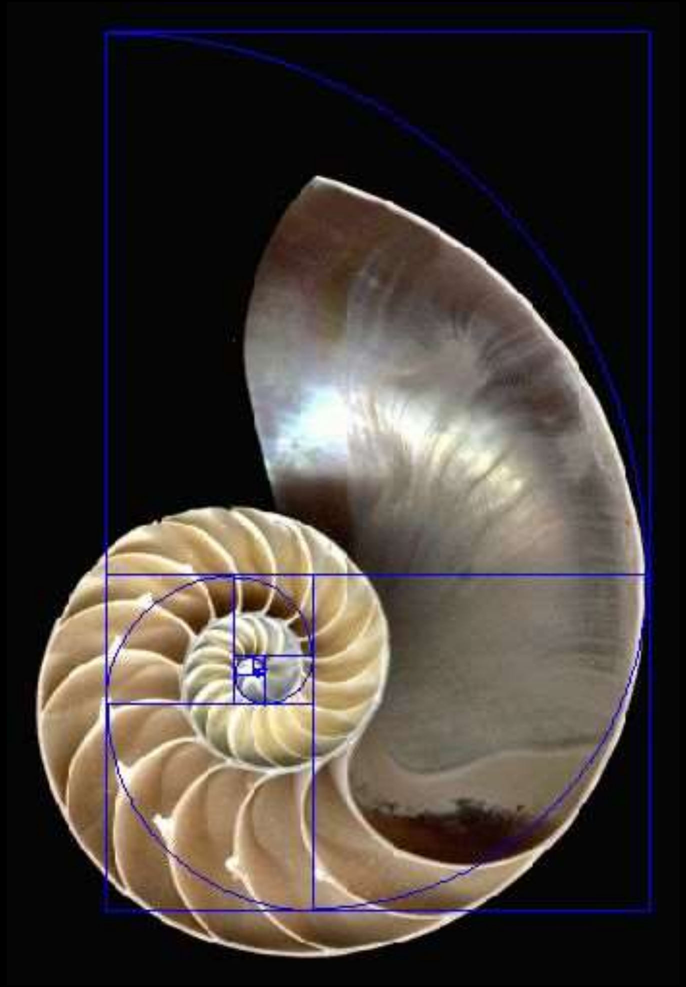
Design Symmetry

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



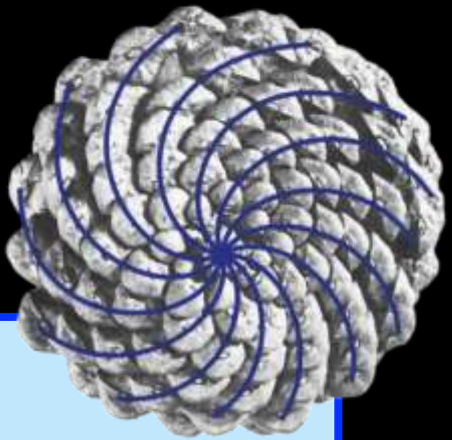
Symmetric:
- Harmonious or Balanced

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

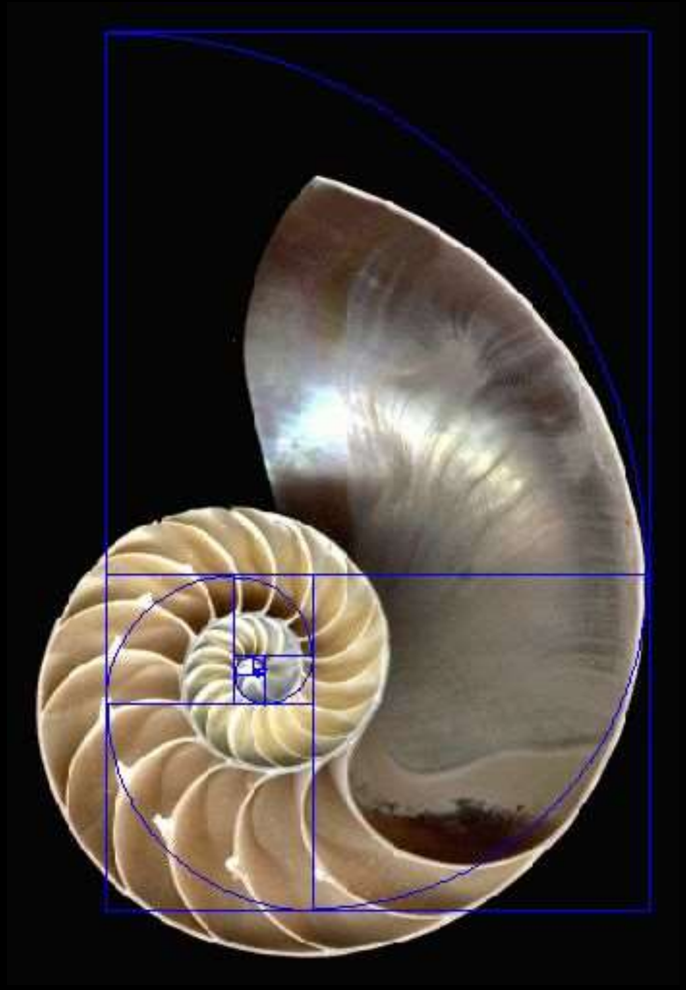


Design Symmetry

<u>Symmetry</u> *(def):*
Agreement in dimensions due to proportion and arrangement



<u>Symmetric:</u>
- Harmonious or Balanced



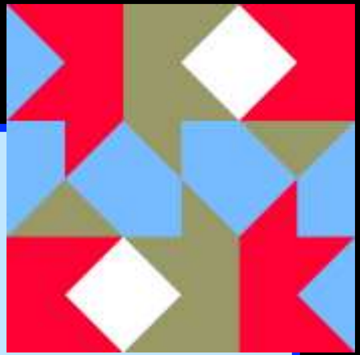
- Q: Why is Symmetry good *(for Design)?*
- A: Symmetry implies **<u>high predictability</u>** and **<u>consistent behavior</u>** *(once pattern is recognized)*

*Enables **<u>system scaling</u>***
(in size and complexity)

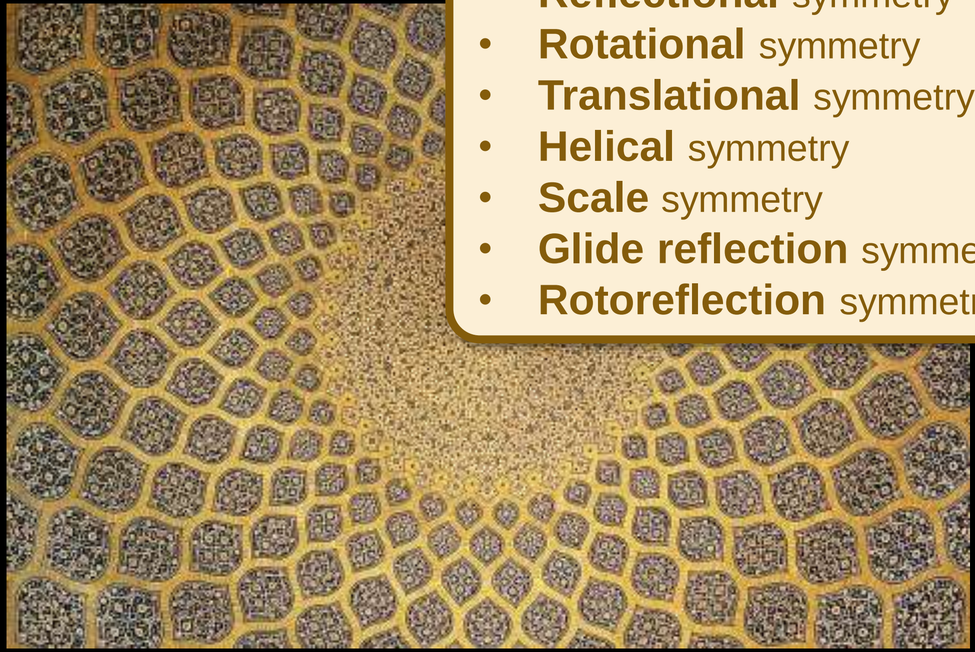
Symmetric Does NOT Mean "Sameness"

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

Symmetric:
- …is "similar"
- …is ***NOT*** "sameness"

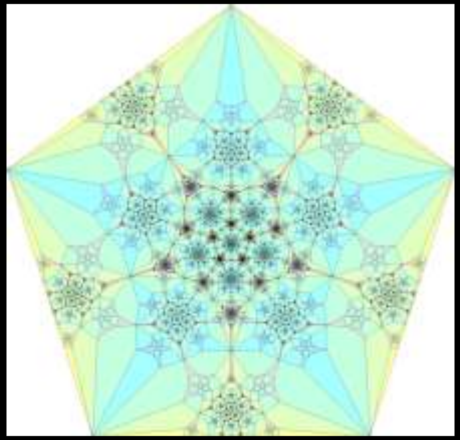


"Kitchen kaleidoscope quilt block"



"The ceiling of Lotfollah mosque, Isfahan, Iran has 8-fold symmetries."

"Celtic knotwork showing p4 symmetry"



"A fractal-like shape that has reflectional symmetry, rotational symmetry and self-similarity"

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

Examples from: https://en.wikipedia.org/wiki/Symmetry

Symmetry Examples

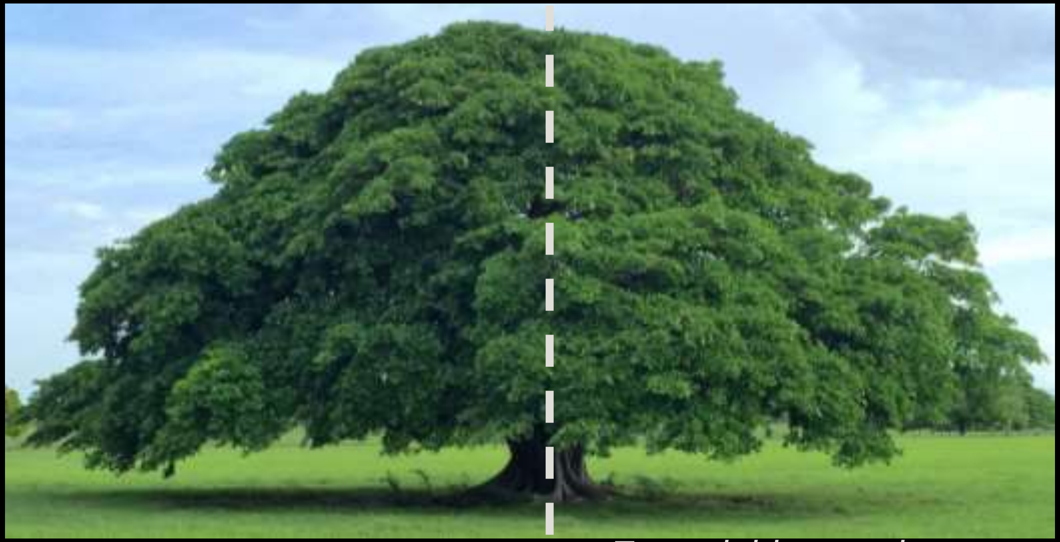
- Q: Guess what is hidden?

Enterolobium cyclocarpum

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

Symmetry Examples

- Q: Guess what is hidden?

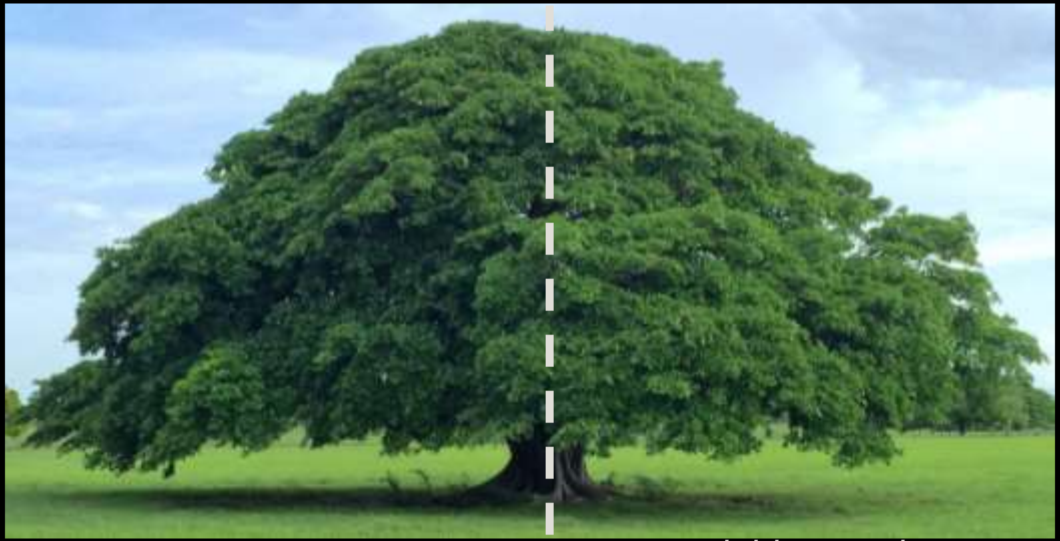


Enterolobium cyclocarpum

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

Symmetry Examples

- Q: Guess what is hidden?

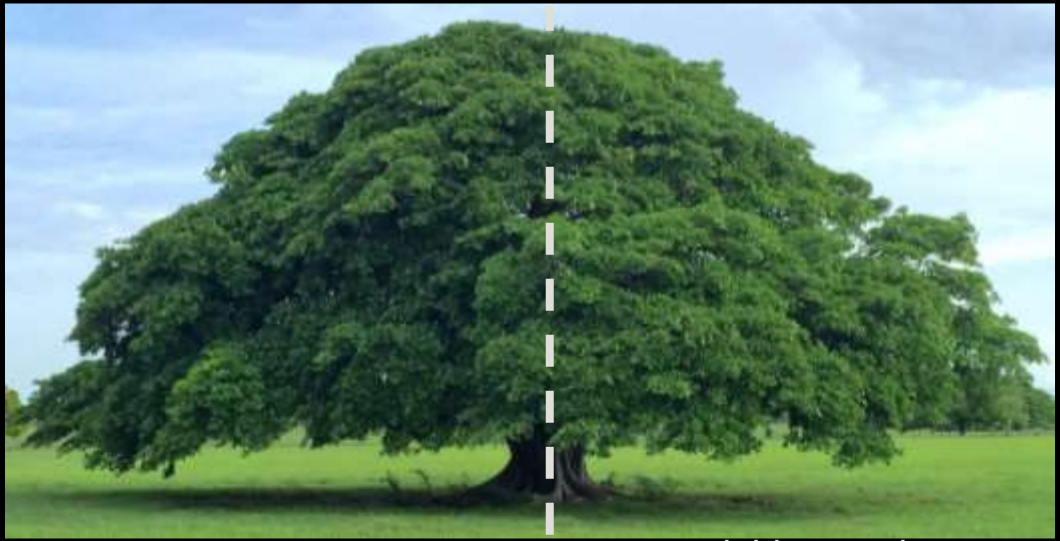


Enterolobium cyclocarpum

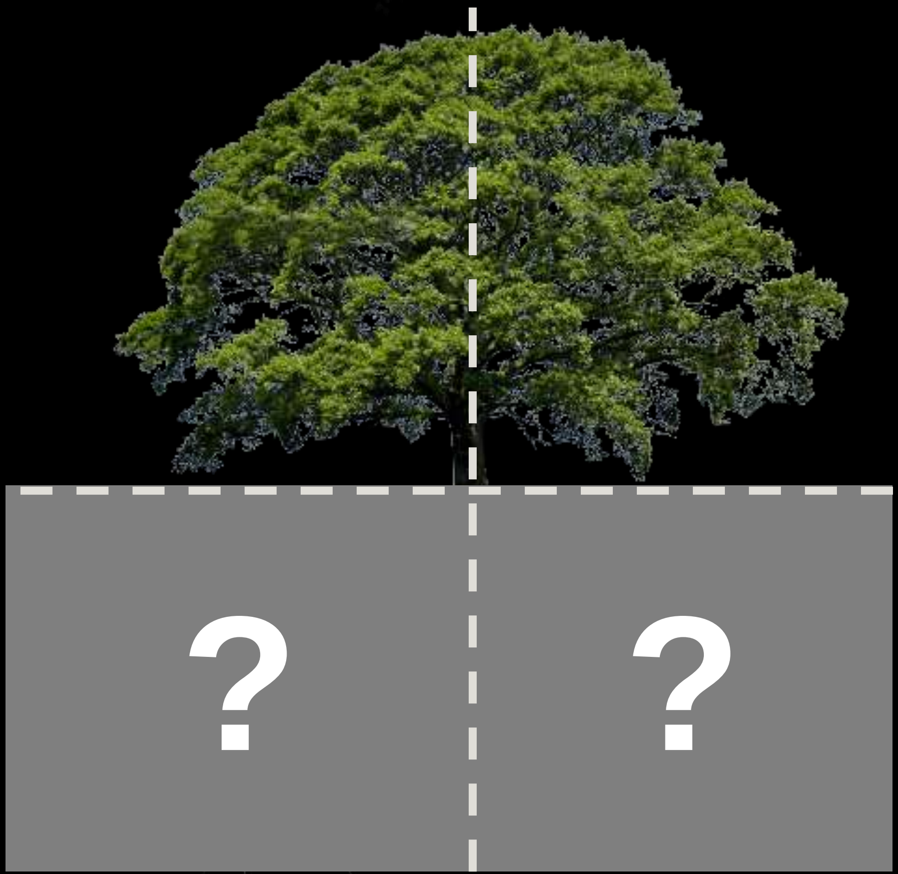
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

Symmetry Examples

- Q: Guess what is hidden?



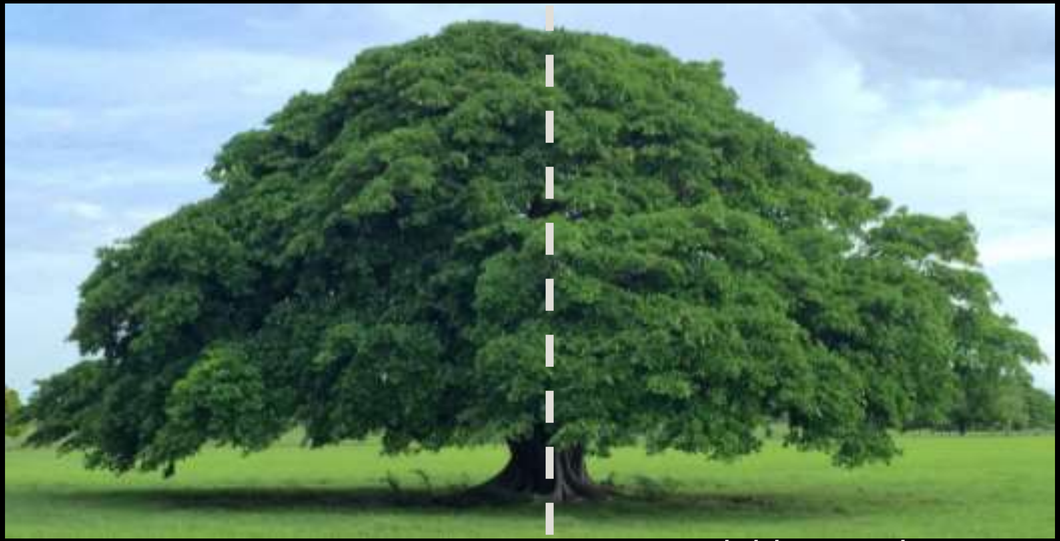
Enterolobium cyclocarpum



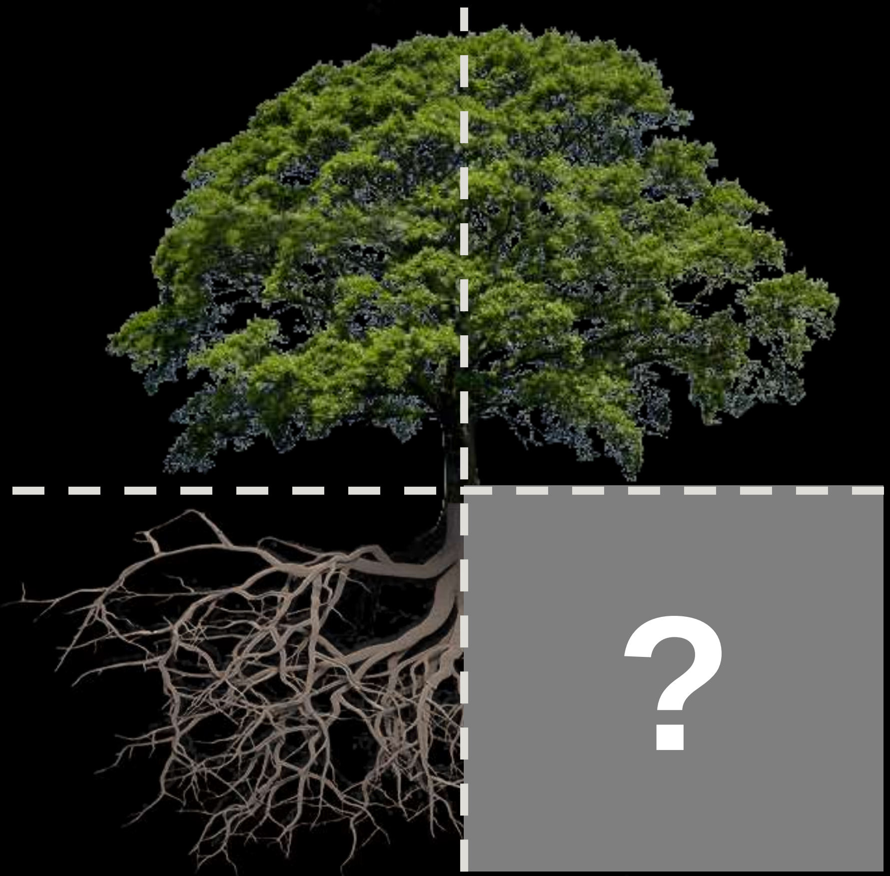
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

Symmetry Examples

- Q: Guess what is hidden?



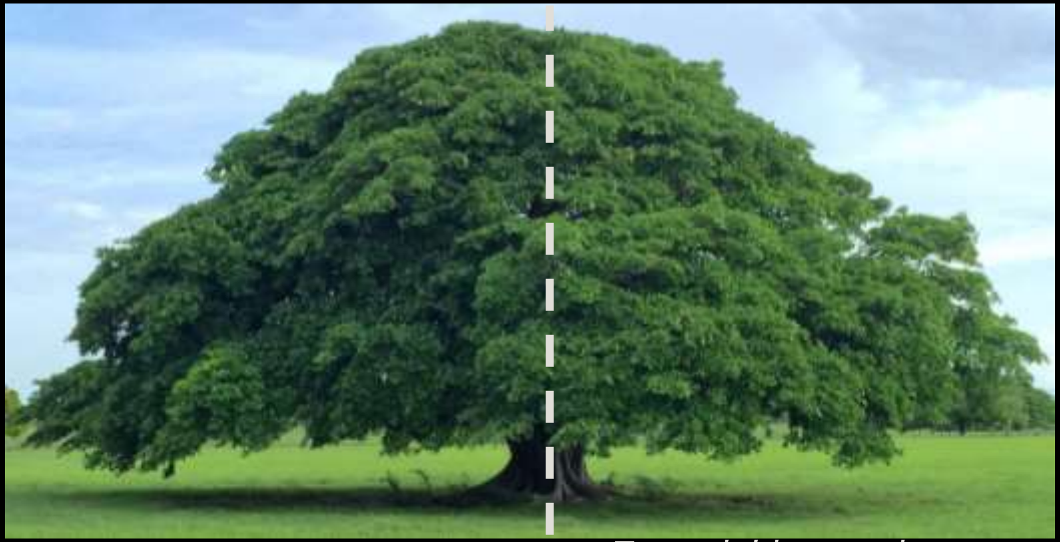
Enterolobium cyclocarpum



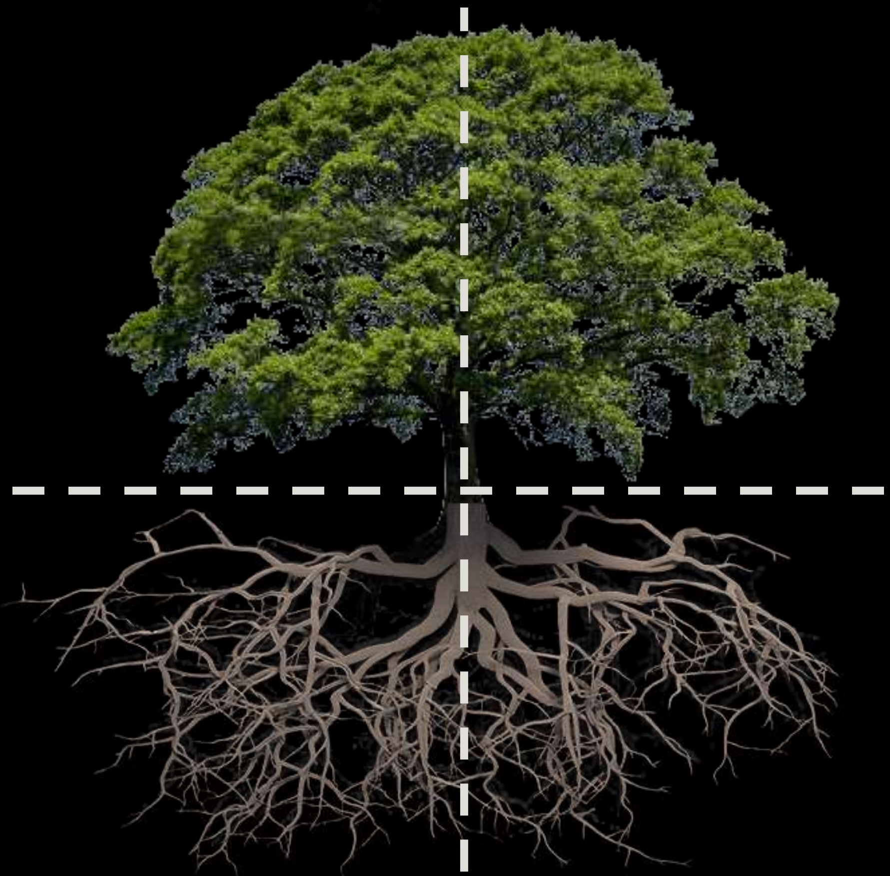
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

Symmetry Examples

- Q: Guess what is hidden?



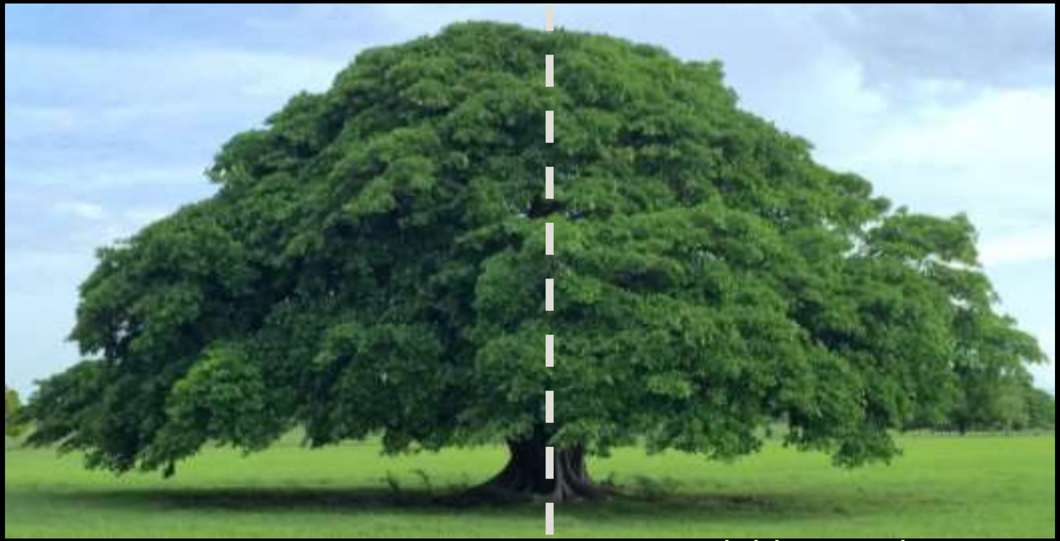
Enterolobium cyclocarpum



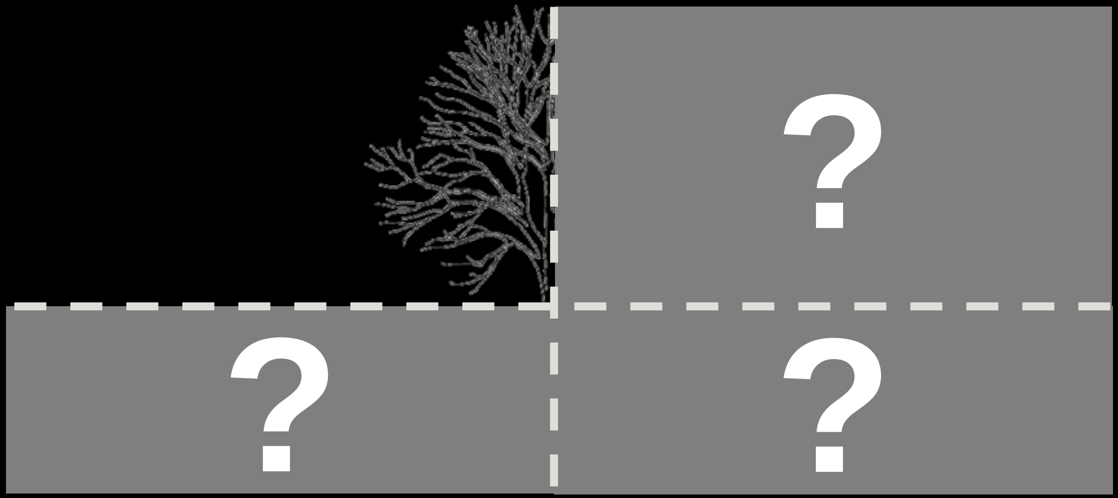
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

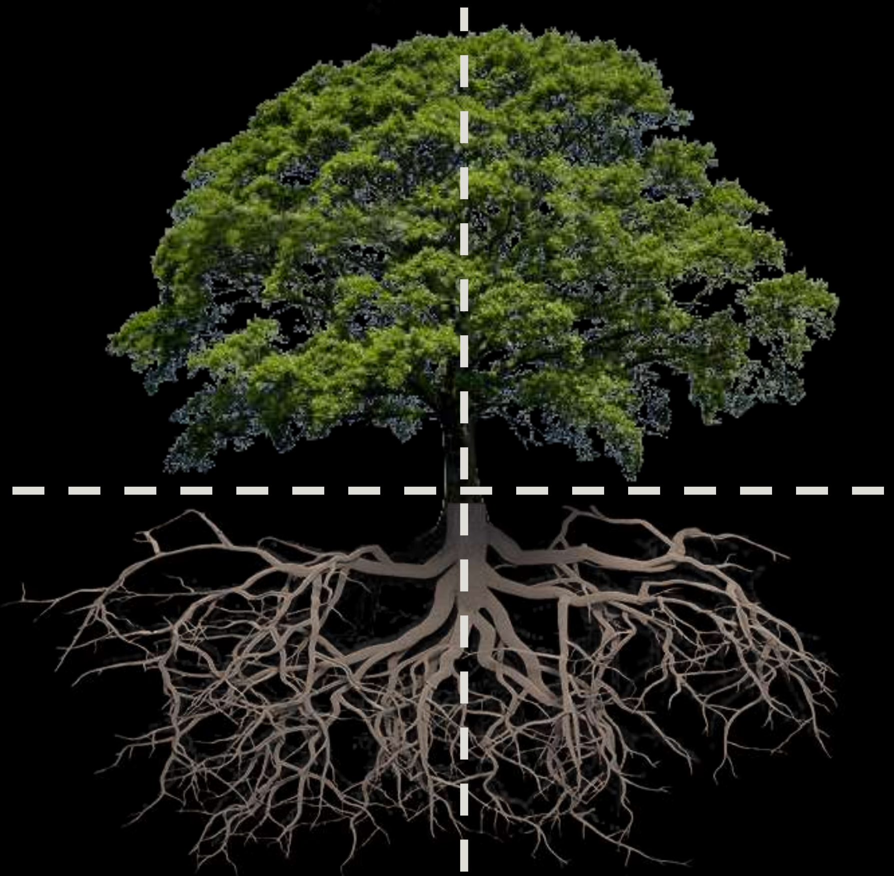
Symmetry Examples

- Q: Guess what is hidden?



Enterolobium cyclocarpum

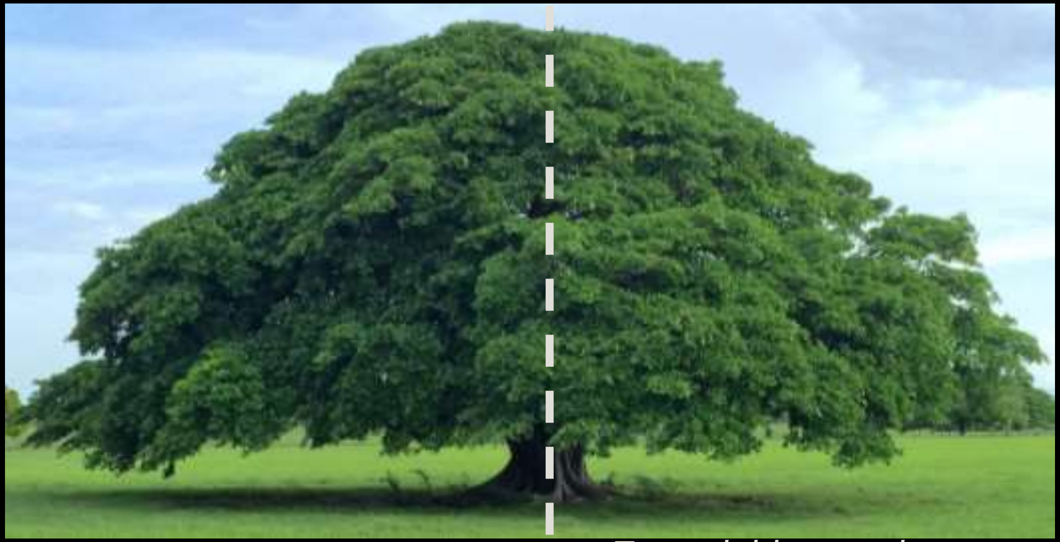




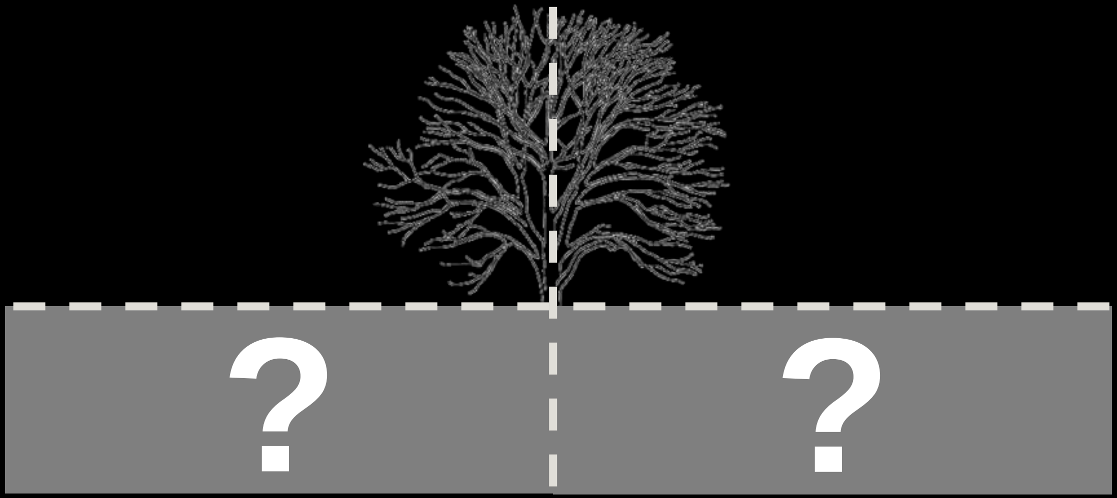
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

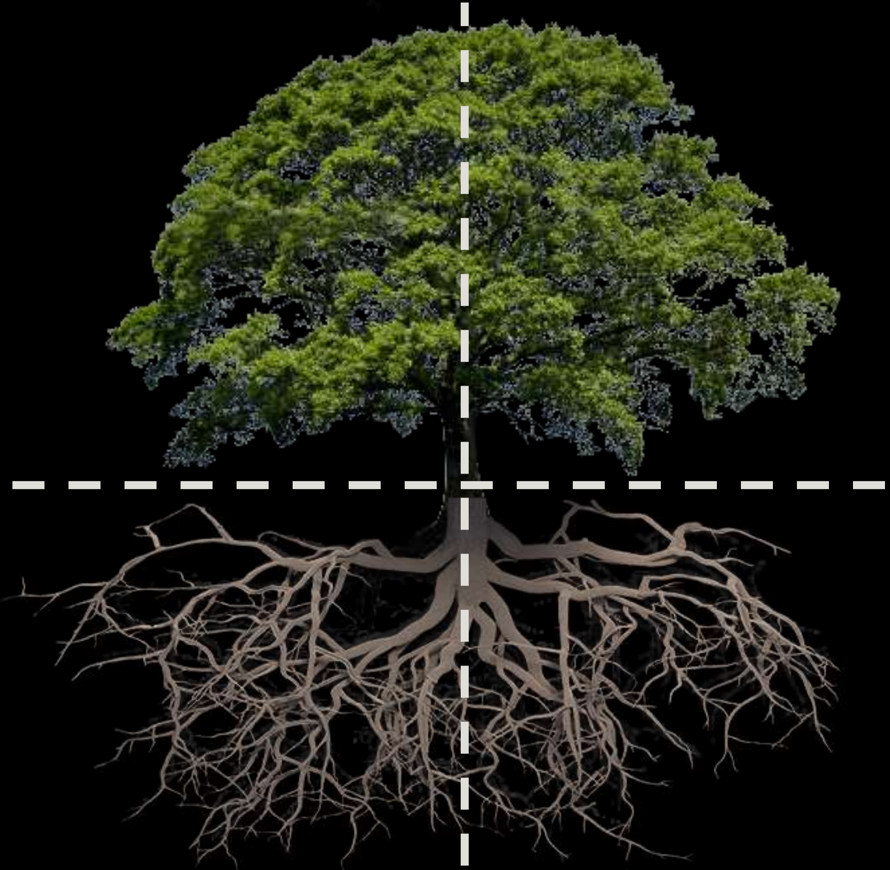
Symmetry Examples

- Q: Guess what is hidden?



Enterolobium cyclocarpum

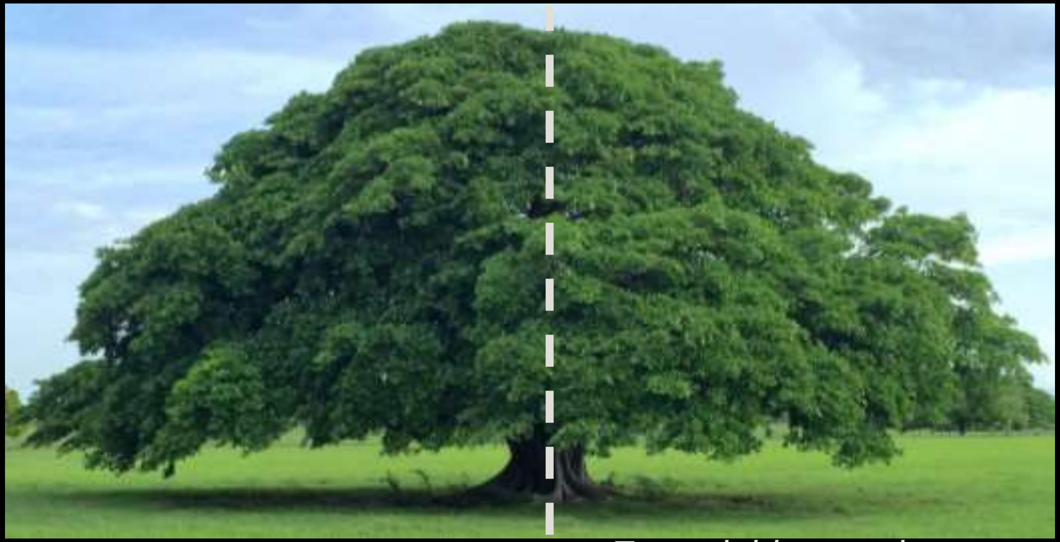




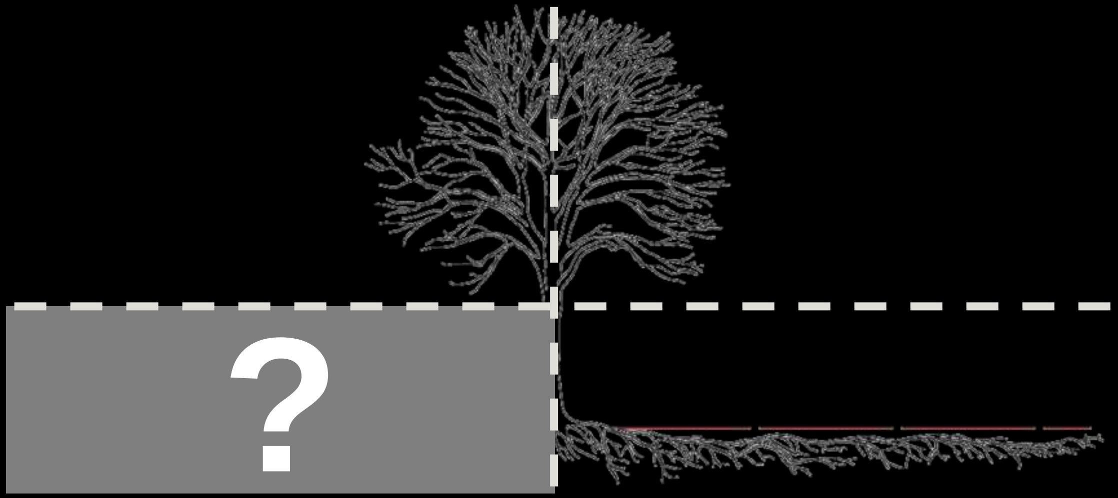
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

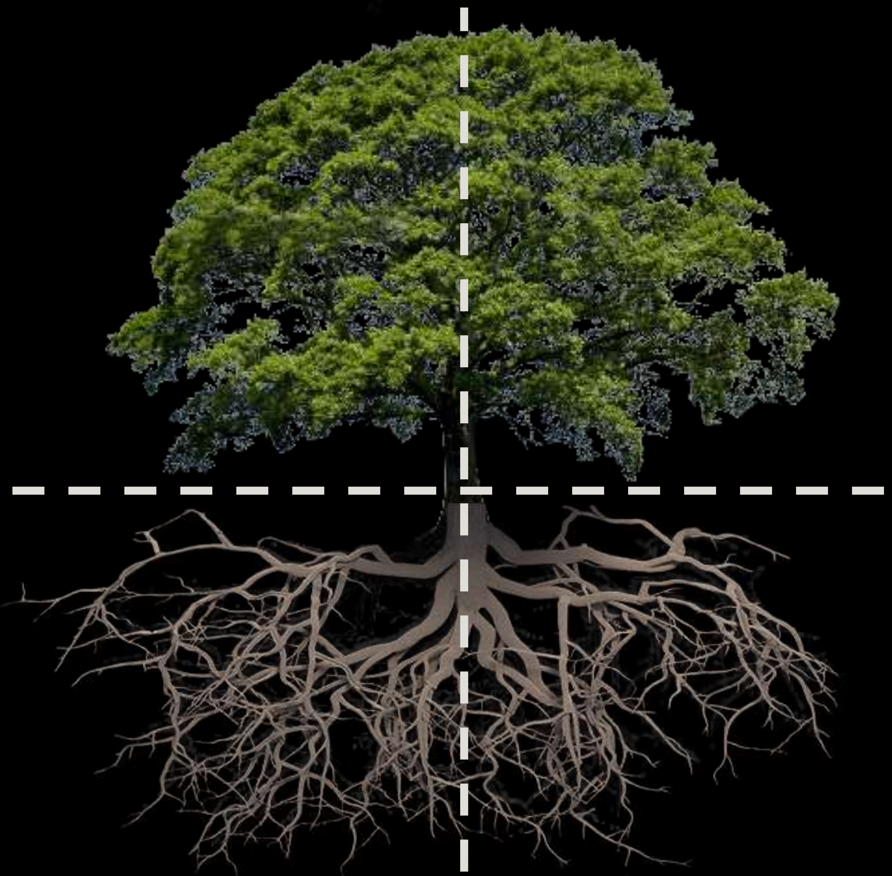
Symmetry Examples

- Q: Guess what is hidden?



Enterolobium cyclocarpum

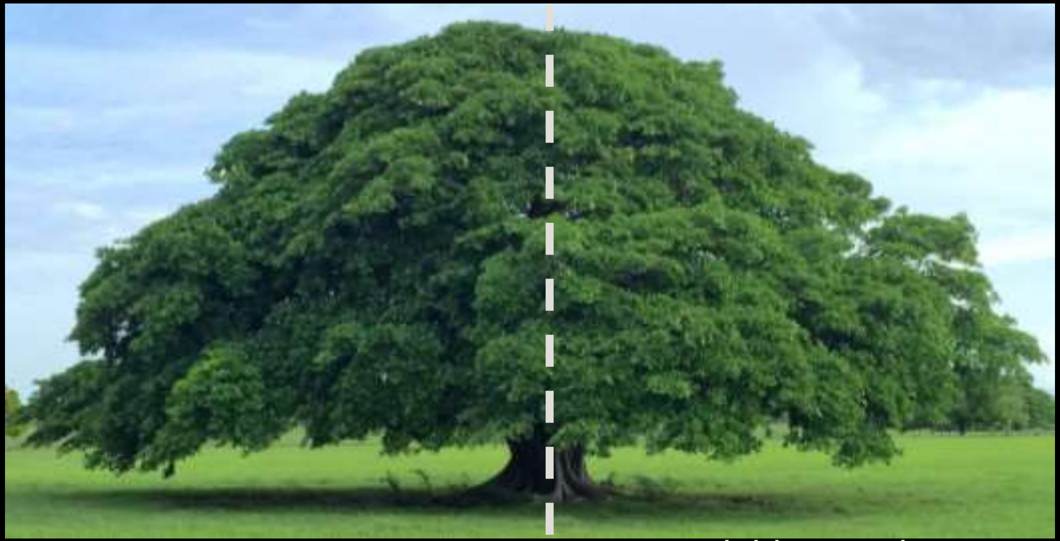




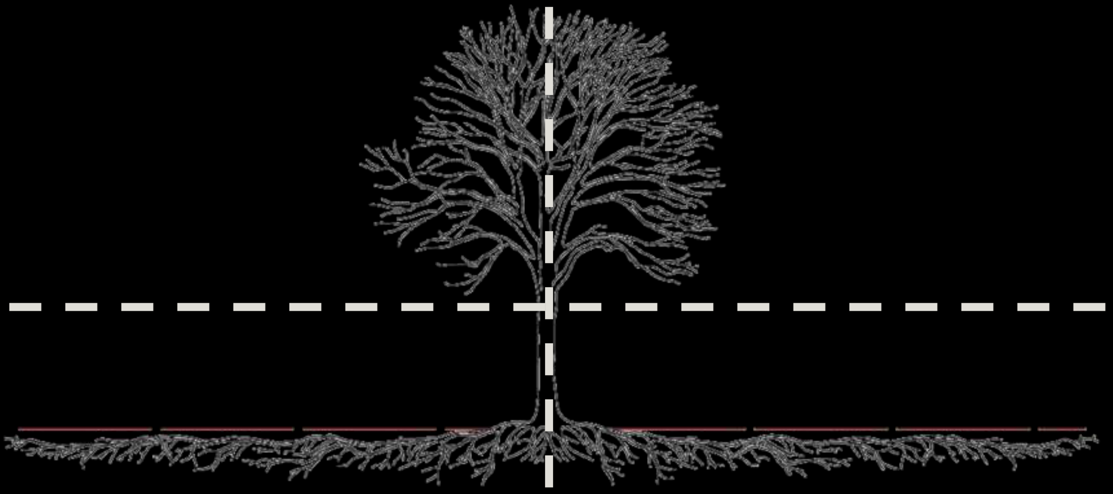
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

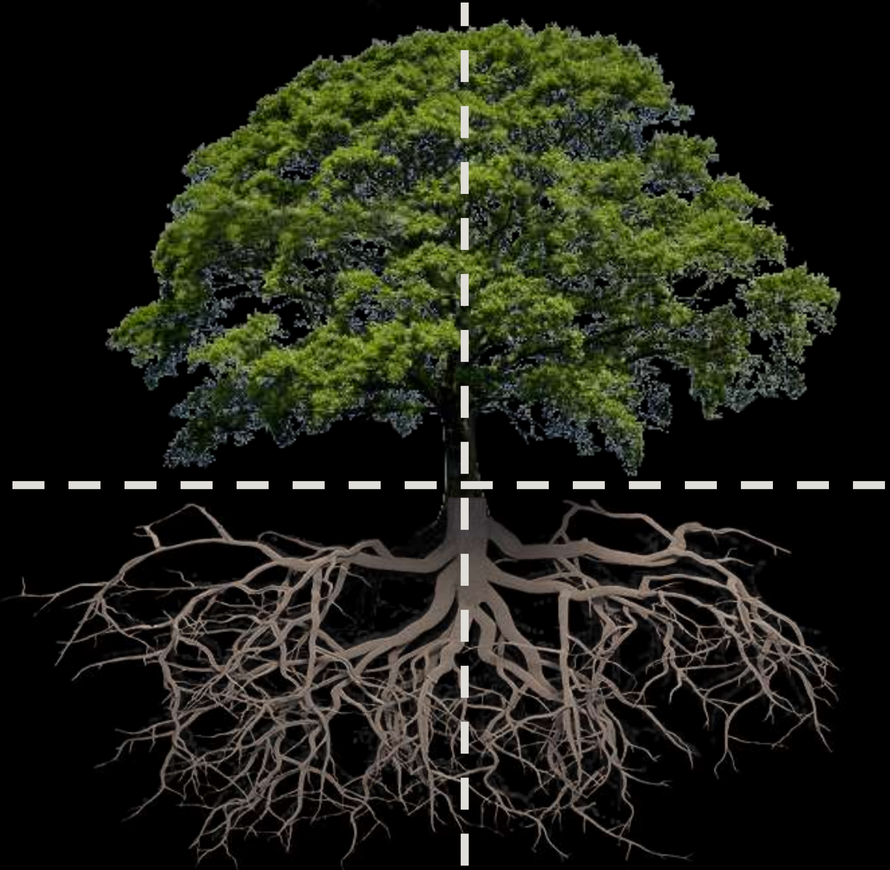
Symmetry Examples

- Q: Guess what is hidden?



Enterolobium cyclocarpum

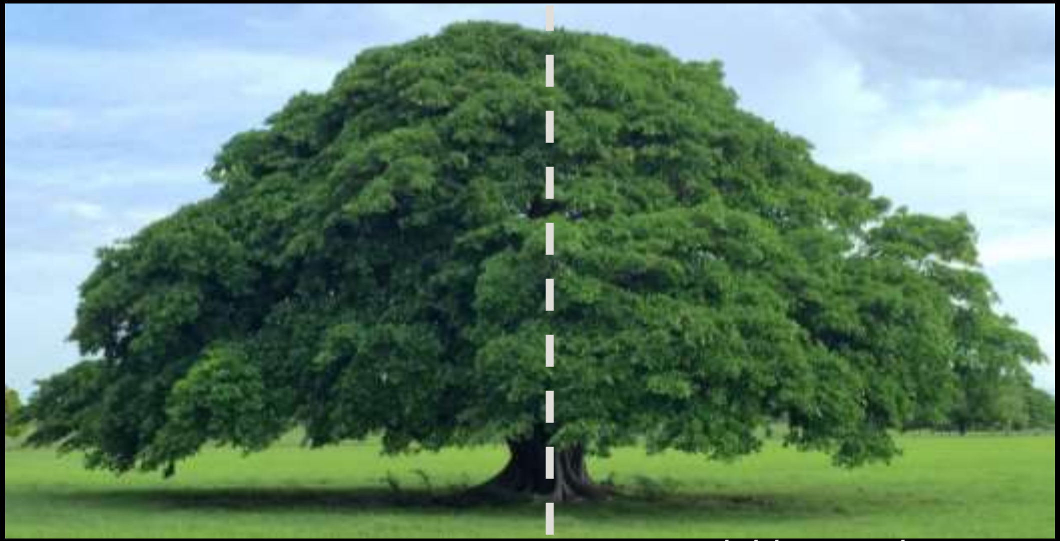




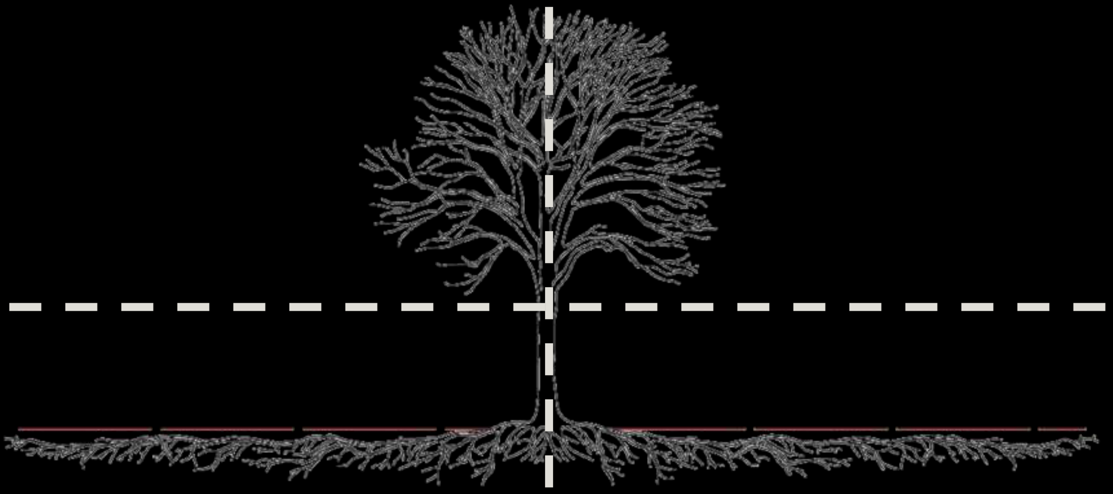
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

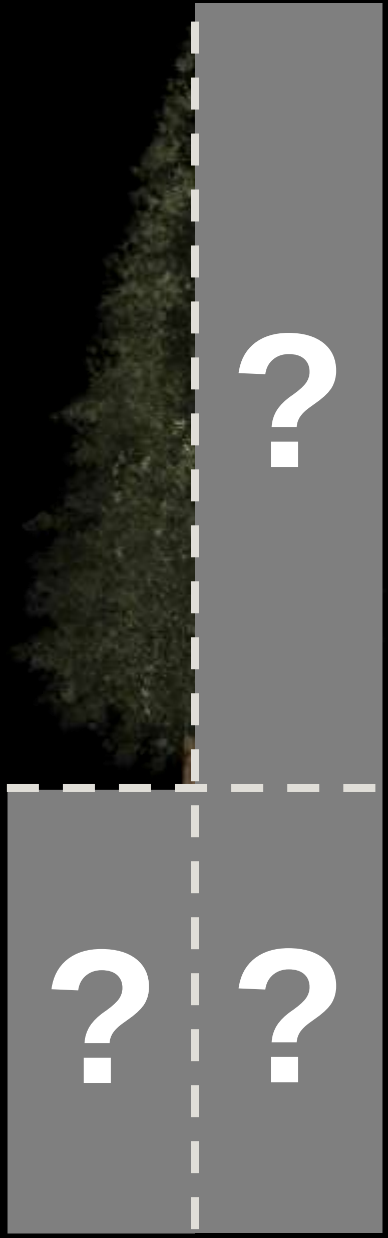
Symmetry Examples

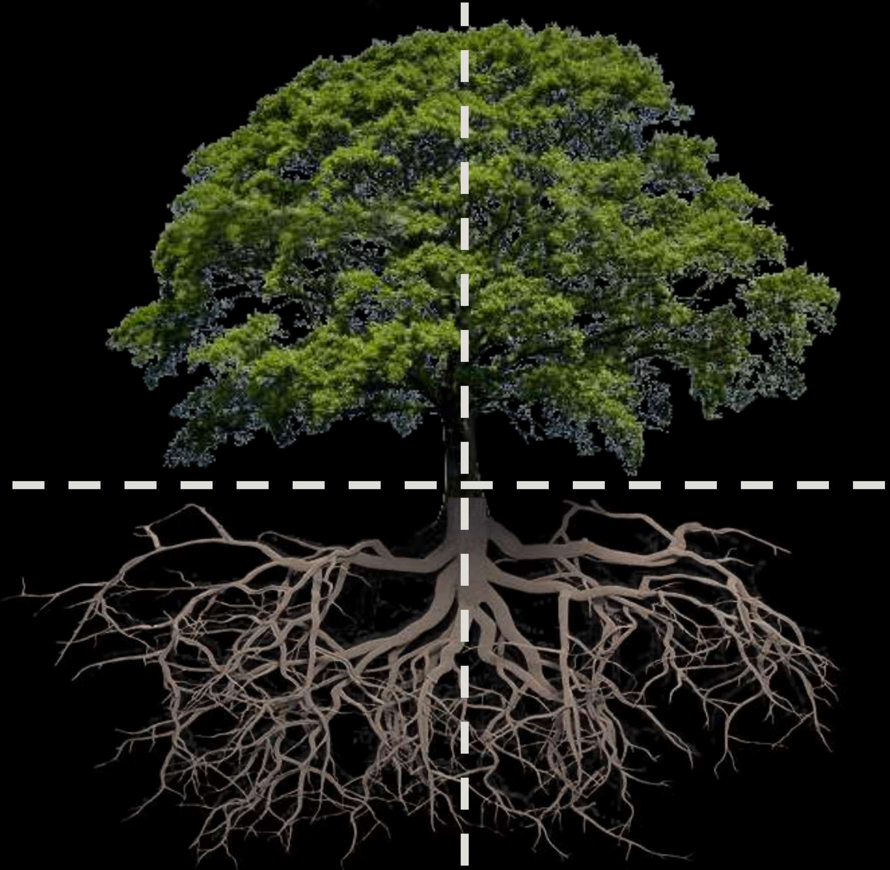
- Q: Guess what is hidden?



Enterolobium cyclocarpum



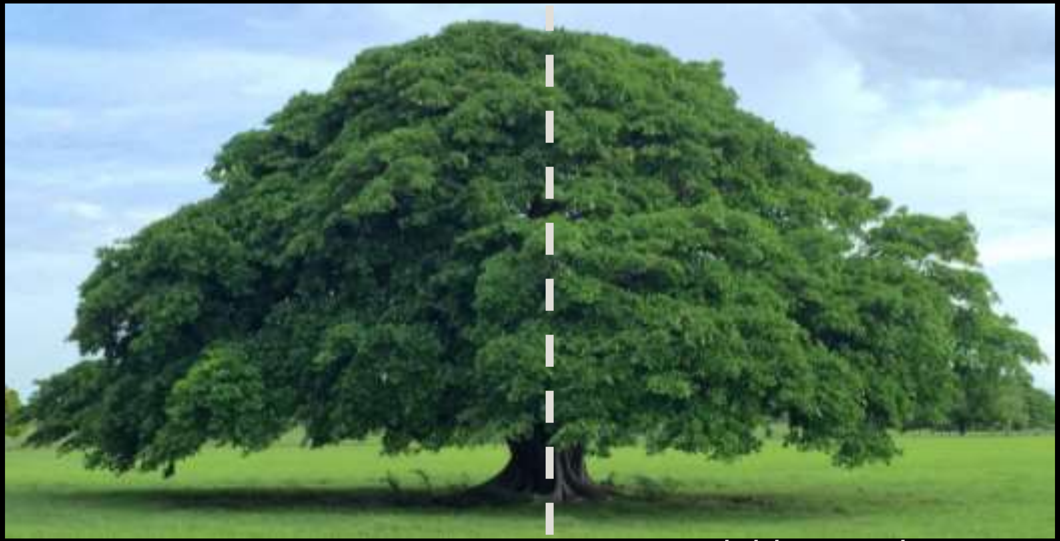




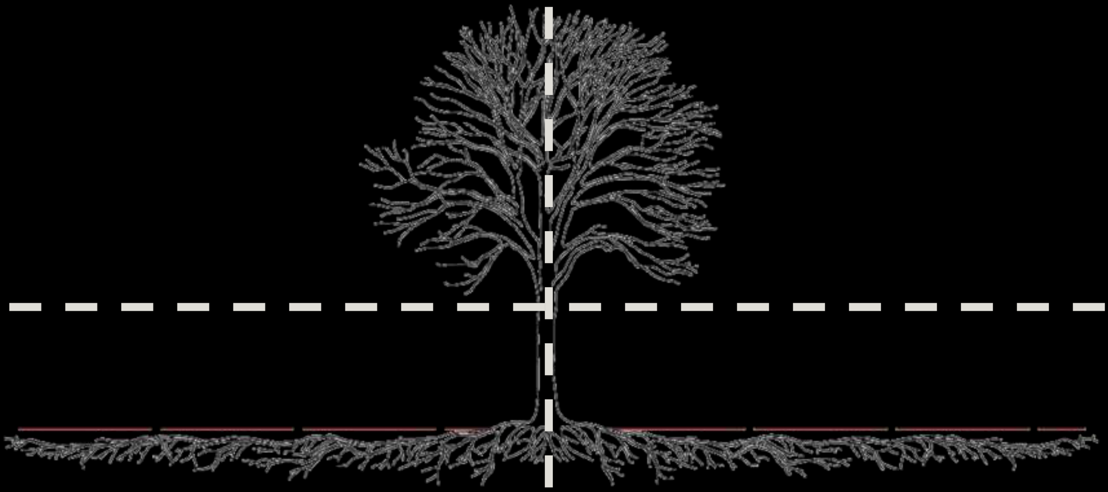
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

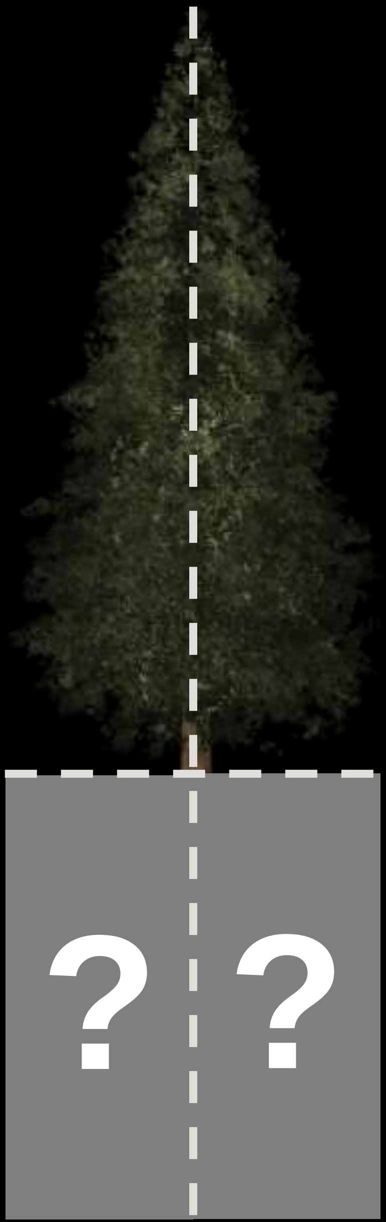
Symmetry Examples

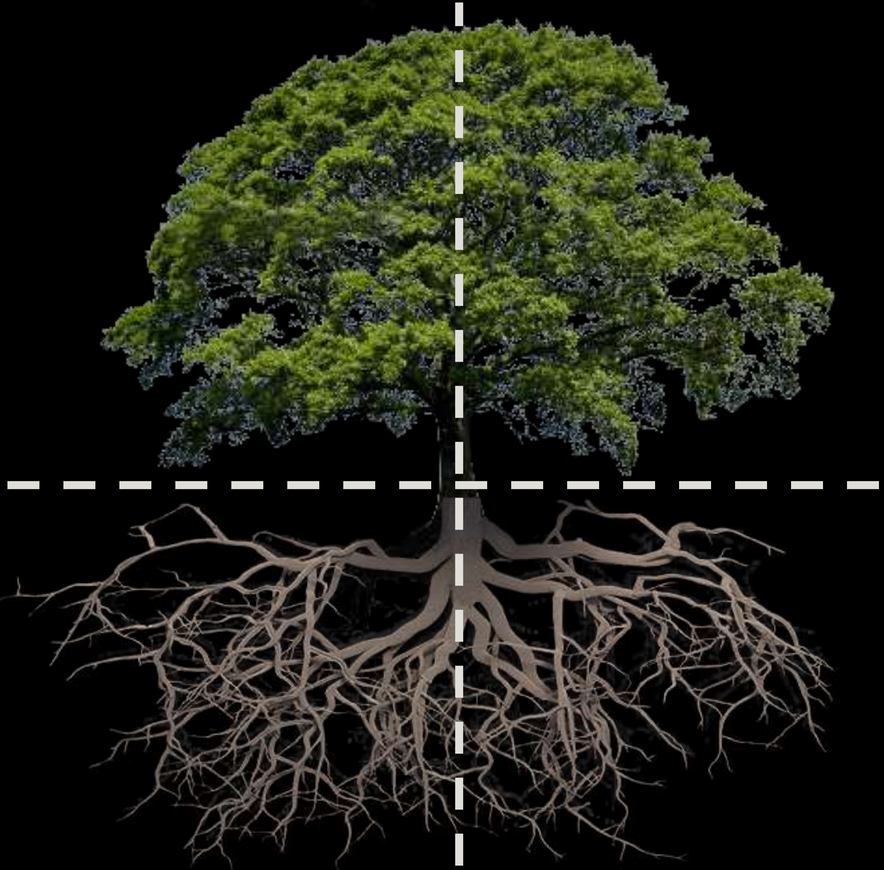
- Q: Guess what is hidden?



Enterolobium cyclocarpum



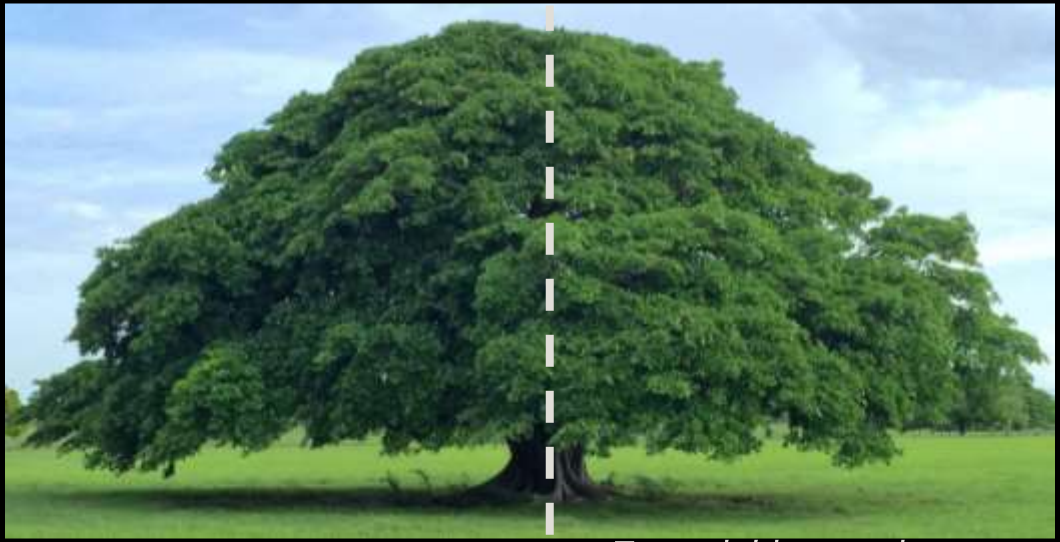




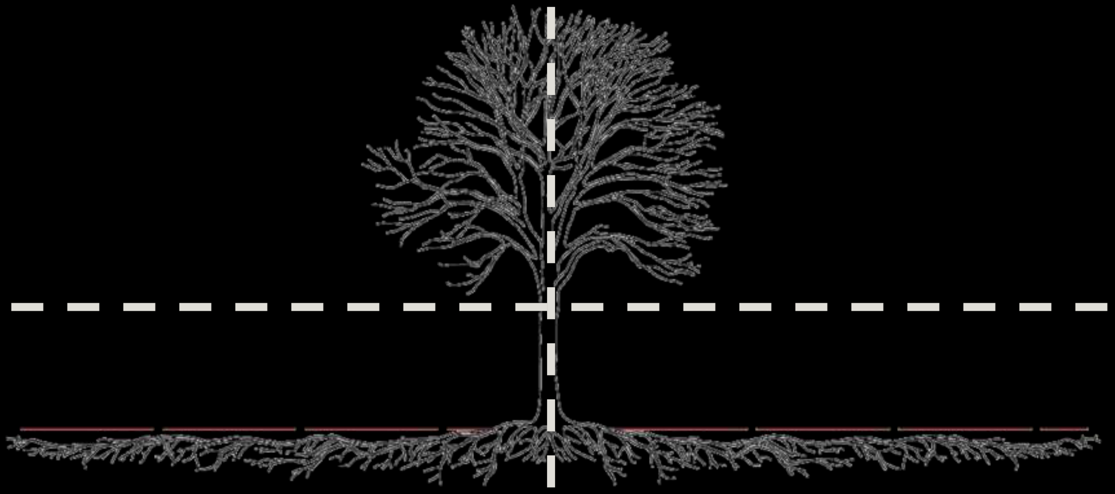
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

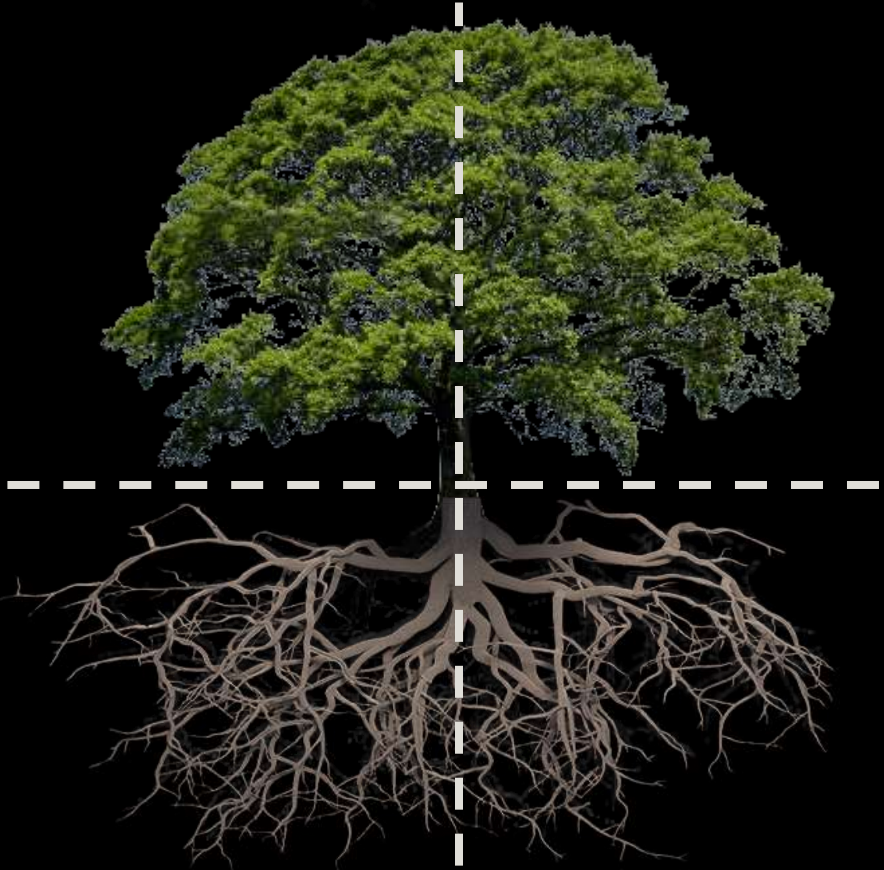
Symmetry Examples

- Q: Guess what is hidden?



Enterolobium cyclocarpum



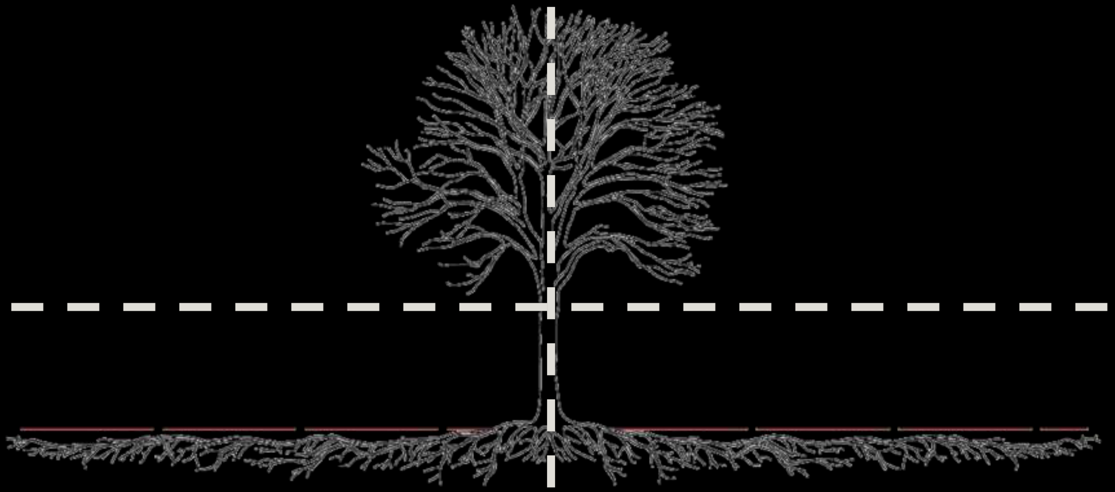


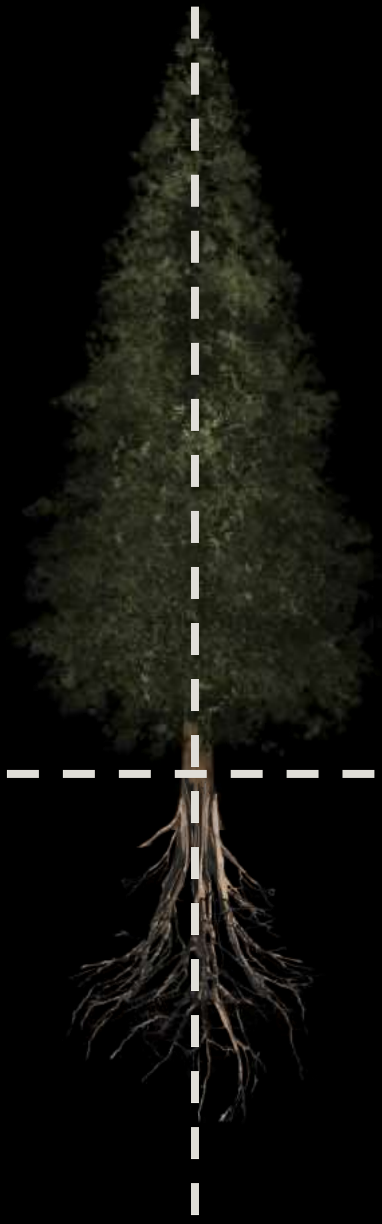
We **use symmetry**
(from what we "see")
to intuit
that which we do not see

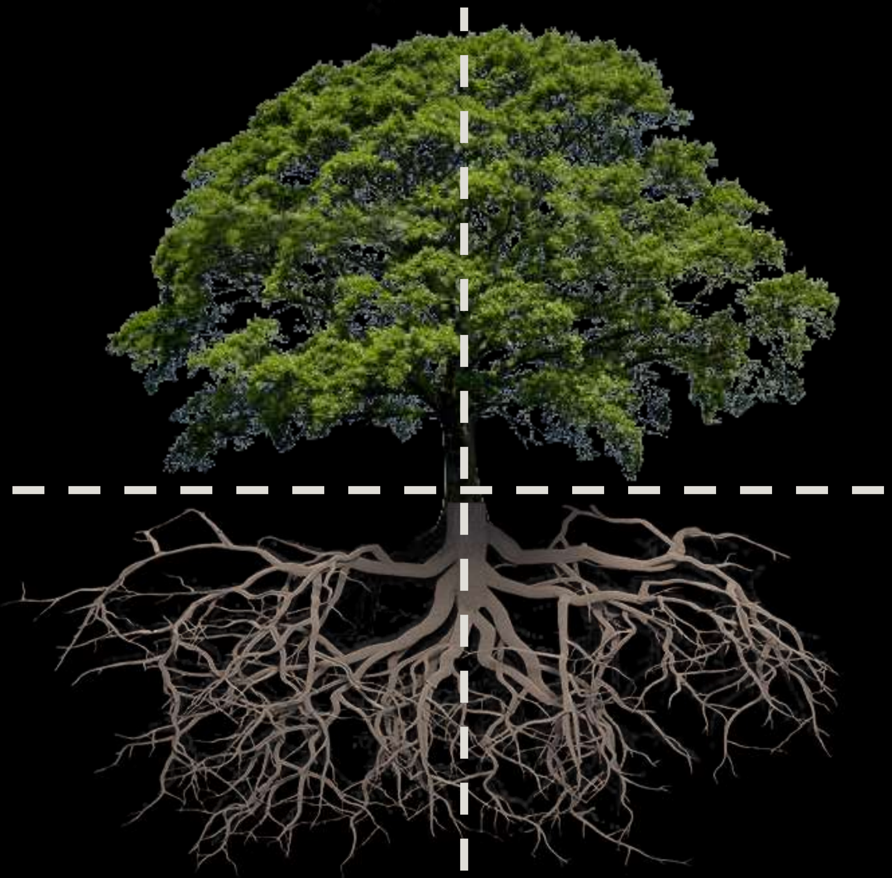
Symmetry Examples

- Q: Guess what is hidden?

Enterolobium cyclocarpum







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

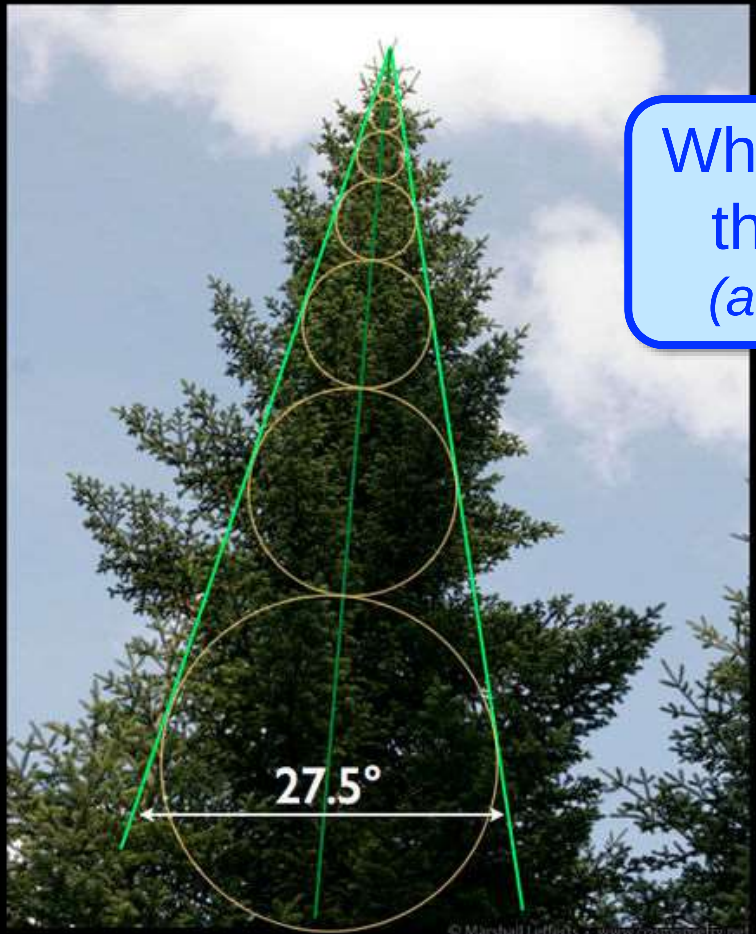
The Phi Scaling Angle In Nature

- Symmetry **naturally occurs** in our system

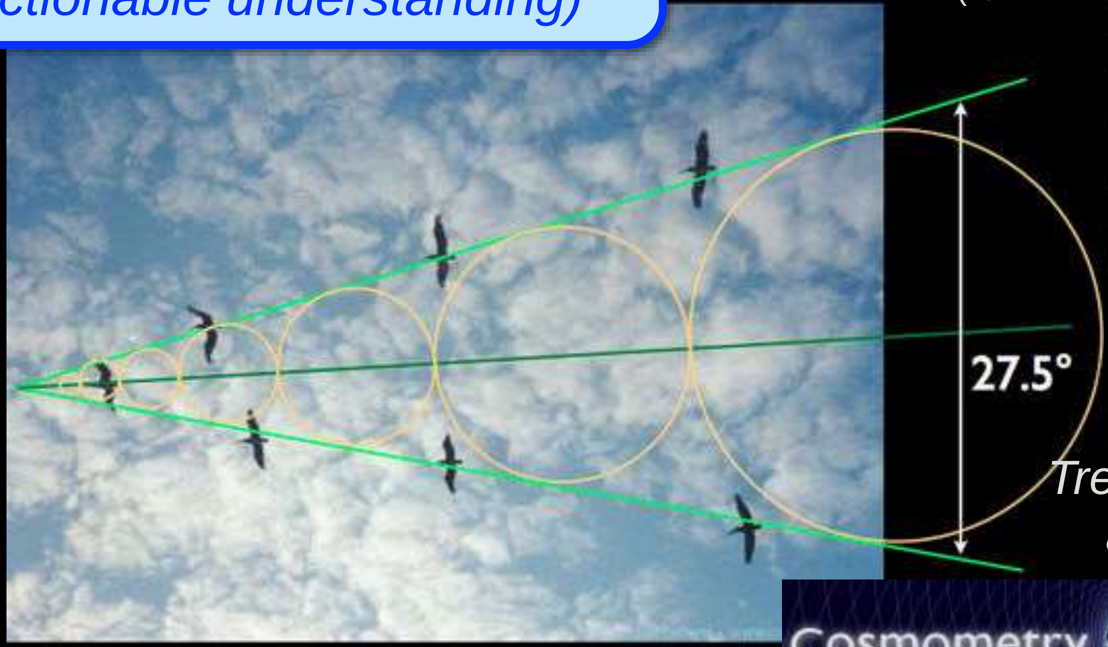
(and in Nature!)

Where there is symmetry, there is **predictability**

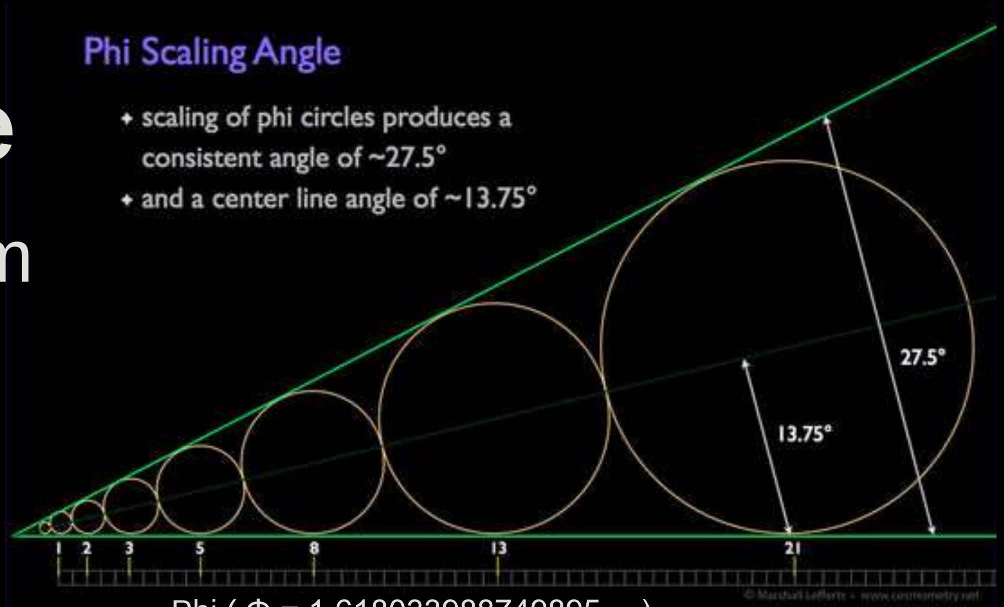
(actionable understanding)



Spruce tree is fractal scaling based on phi angle

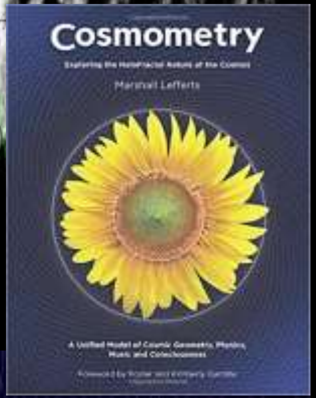


Pelicans in flight with phi angle



Phi (Φ = 1.618033988749895…)

Tree bark scaling at phi angle



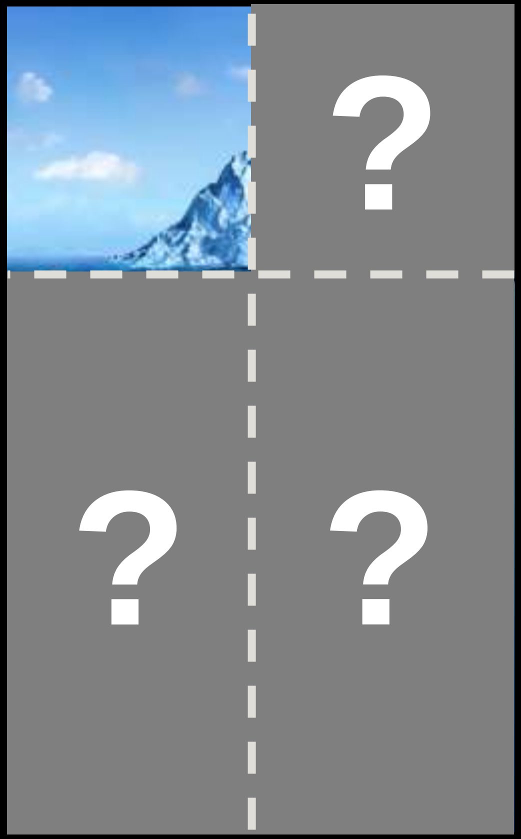
https://cosmometry.net/phi-scaling-angle

Symmetry allows us to “**<u>know” things</u>** 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)*

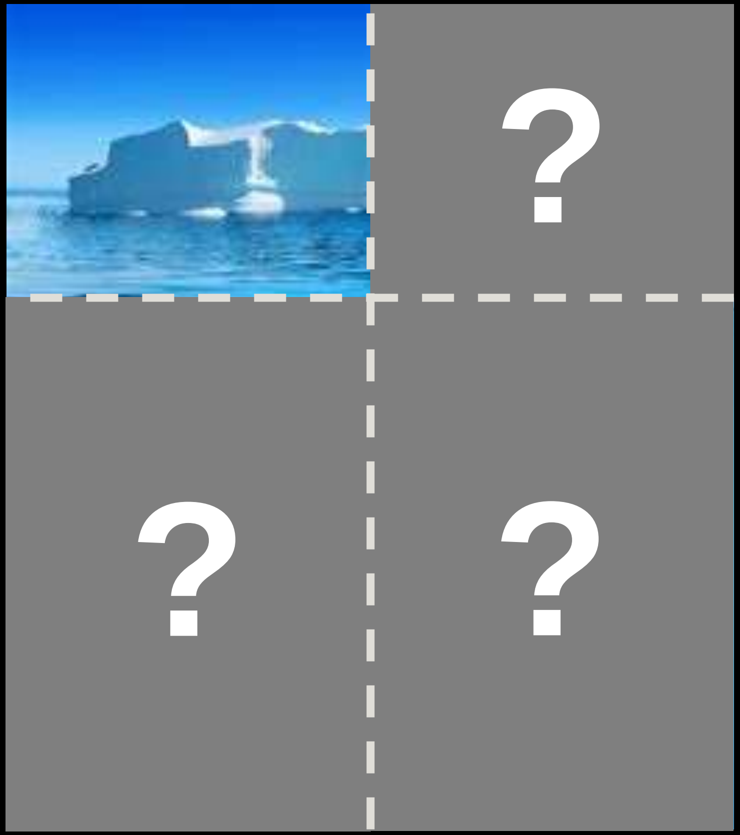
- Q: Guess what is hidden?

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?

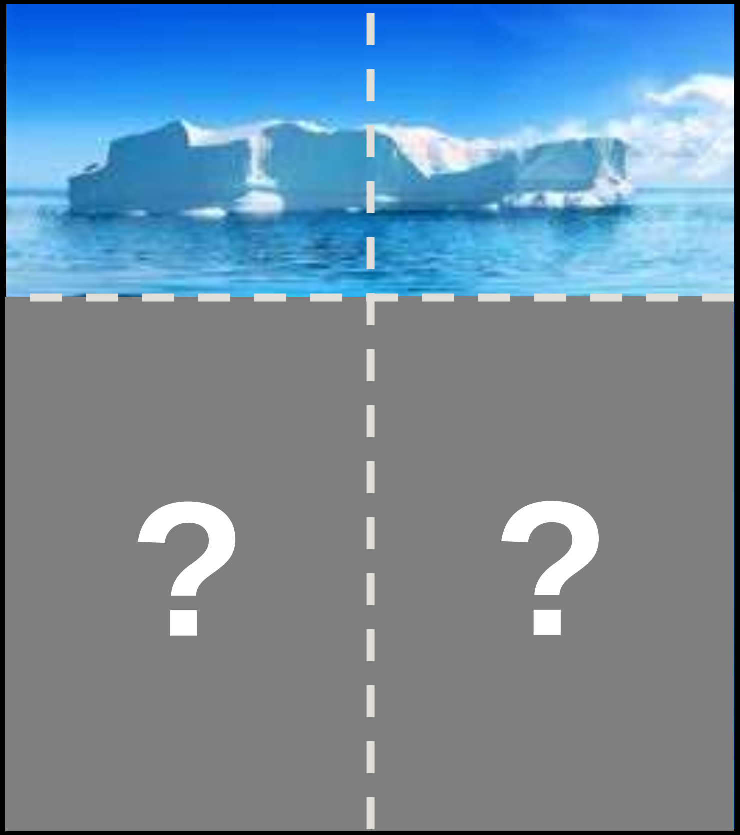


We **use symmetry** *(from what we "see")* **to intuit** that which we do not see

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**

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**

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"

```
...
{
  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"

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

Enter the block, object is created

Leave the block, object is destroyed

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

```
{
  ...
  Bar* b = new Bar();
  ConsumeBar(*b);
}
```

```
void ConsumeBar(Bar& b) {
  ProcessBar(b);
  delete b; //...consume
}
```

Symmetry of the C++ Stack

"Scope Symmetry"

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

Use whenever possible!

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


    //...do stuff with b0
}
```

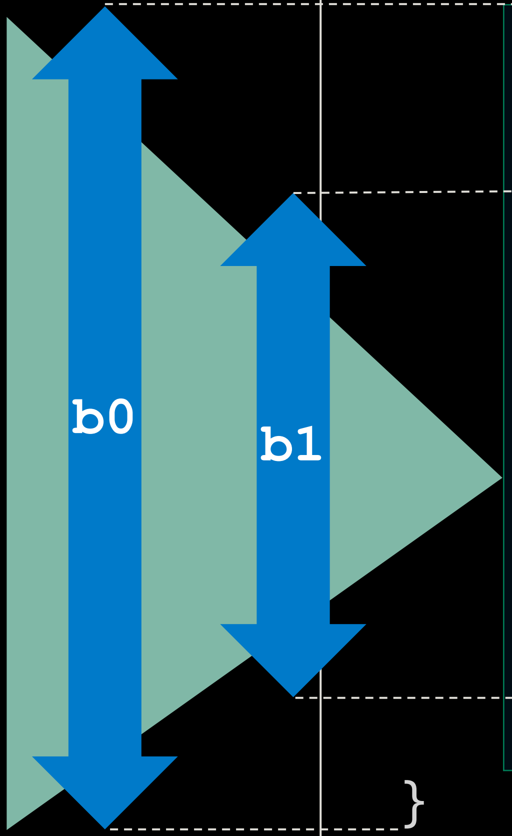
b0

Symmetry of the C++ Stack

"Scope Symmetry"

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

Use whenever possible!



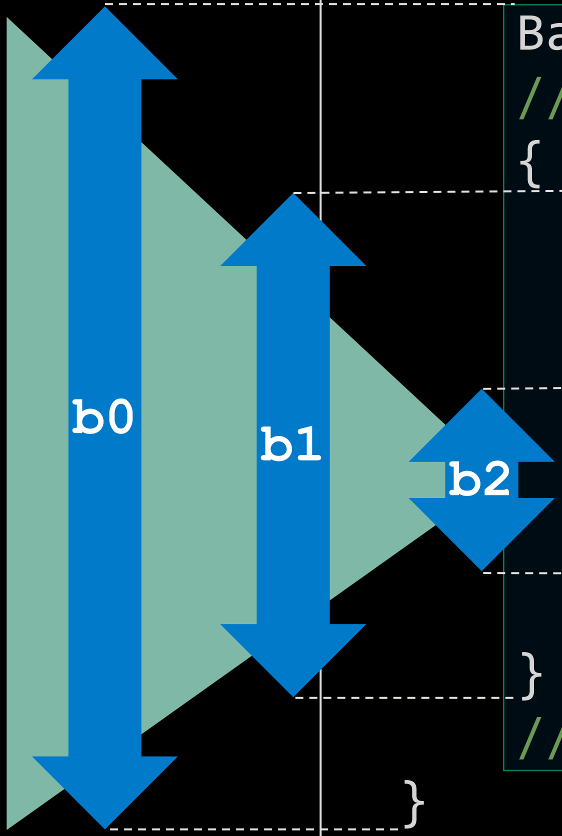

```
...
{
    Bar b0;
    //...do stuff with b0
    {
        Bar b1;
        //...do stuff with b0, b1



        //...do stuff with b0, b1
    }
    //...do stuff with b0
}
```

Symmetry of the C++ Stack

"Scope Symmetry"

Stack-based objects have
<u>outstanding lifecycle symmetry</u>
(Guaranteed!)

Use whenever possible!



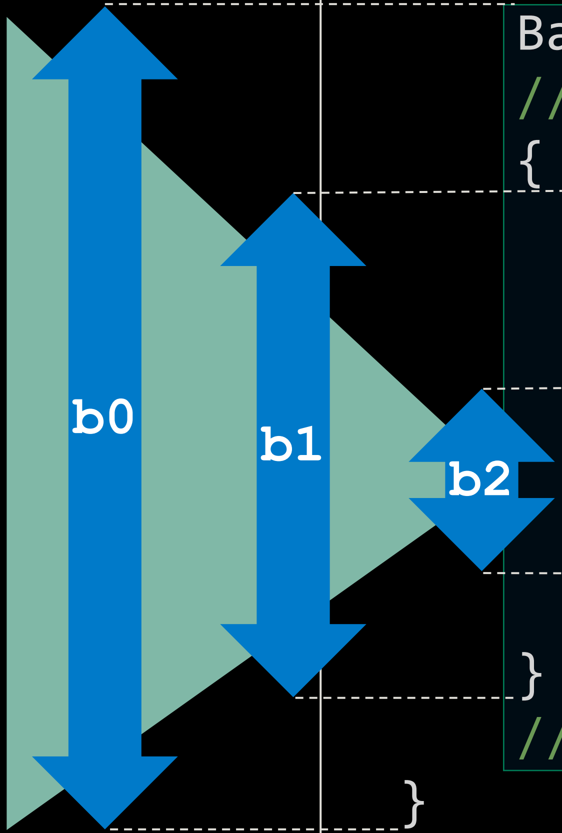

```
...
{
    Bar b0;
    //...do stuff with b0
    {
        Bar b1;
        //...do stuff with b0, b1
        {
            Bar b2;
            //...do stuff with b0, b1, b2
        }
        //...do stuff with b0, b1
    }
    //...do stuff with b0
}
```

Symmetry of the C++ Stack

"Scope Symmetry"

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

Use whenever possible!




```
...
{
    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
}
```

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

Symmetry of the C++ Heap

```
{
  ...
  {
    Bar* b = new Bar();
    //...do stuff with b0
    {
      //...arbitrary code
      DoThing0(*b);
      DoThing1(*b);
    }
    ConsumeBar(*b);
  }
}
```

Bar()…~Bar()

```
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
}
```

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

This is a (useful!) ***design feature***

Symmetry of the C++ Heap

```
{
  ...
  {
    Bar* b = new Bar();
    //...do stuff with b0
    {
      //...arbitrary code
      DoThing0(*b);
      DoThing1(*b);
    }
    ConsumeBar(*b);
  }
}
```

`Bar()…~Bar()`

```
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
}
```

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

This is a (useful!) ***design feature***

Your Design defines object lifecycle

- **`std::unique<>`** is an implementation tool, *not* a design decision

Symmetry of the C++ Heap

```
{
  ...
  {
    Bar* b = new Bar();
    //...do stuff with b0
    {
      //...arbitrary code
      DoThing0(*b);
      DoThing1(*b);
    }
    ConsumeBar(*b);
  }
}
```

Bar()…~Bar()

```
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
}
```

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)*:

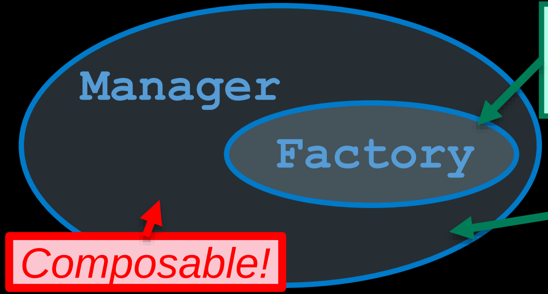
(Strong!) Object Lifecycle Symmetry

C	C++
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

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

- We use "Design" to **establish Symmetry**
  - *Example:* Restore Symmetry when using **`new`**…**`delete`**



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!

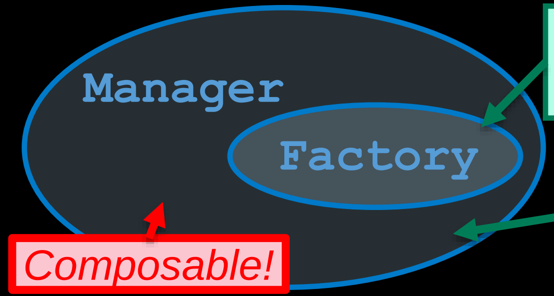
`Manager` type provides options for:

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

Use Design To Establish Symmetry

These are
generalized patterns
(variations may apply)

- We use "Design" to **establish Symmetry**
  - *Example:* Restore Symmetry when using `new`…`delete`



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:

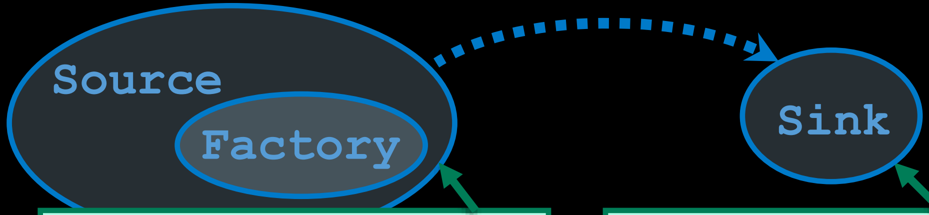
- No leaked instances
- Allocation amortization
- Whole-system reset
- Resource prioritization and recovery
- Resource metrics and runtime monitoring

Source…Sink:

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

Symmetry Restored!

Resource lifecycle is **well-defined**



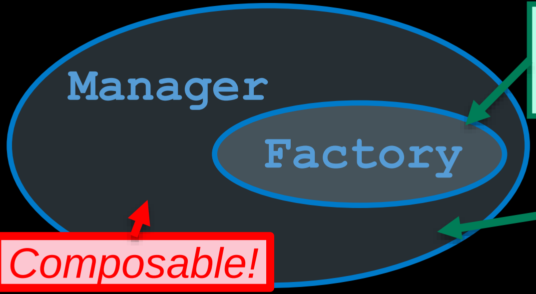
Performs `new` *(and ownership transfer)*

Performs `delete` after instance processing

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)



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!

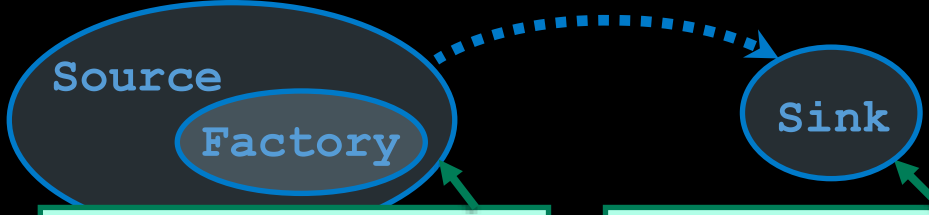
Symmetry Restored!
`Manager` type provides options for:

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

Source…Sink:

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

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

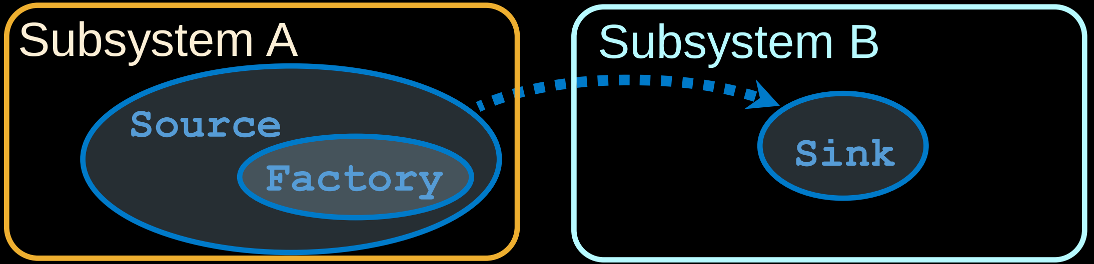


Performs `new`
(and ownership transfer)

Performs `delete` after instance processing

Producer…Consumer:

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



Expressing Symmetry Through API

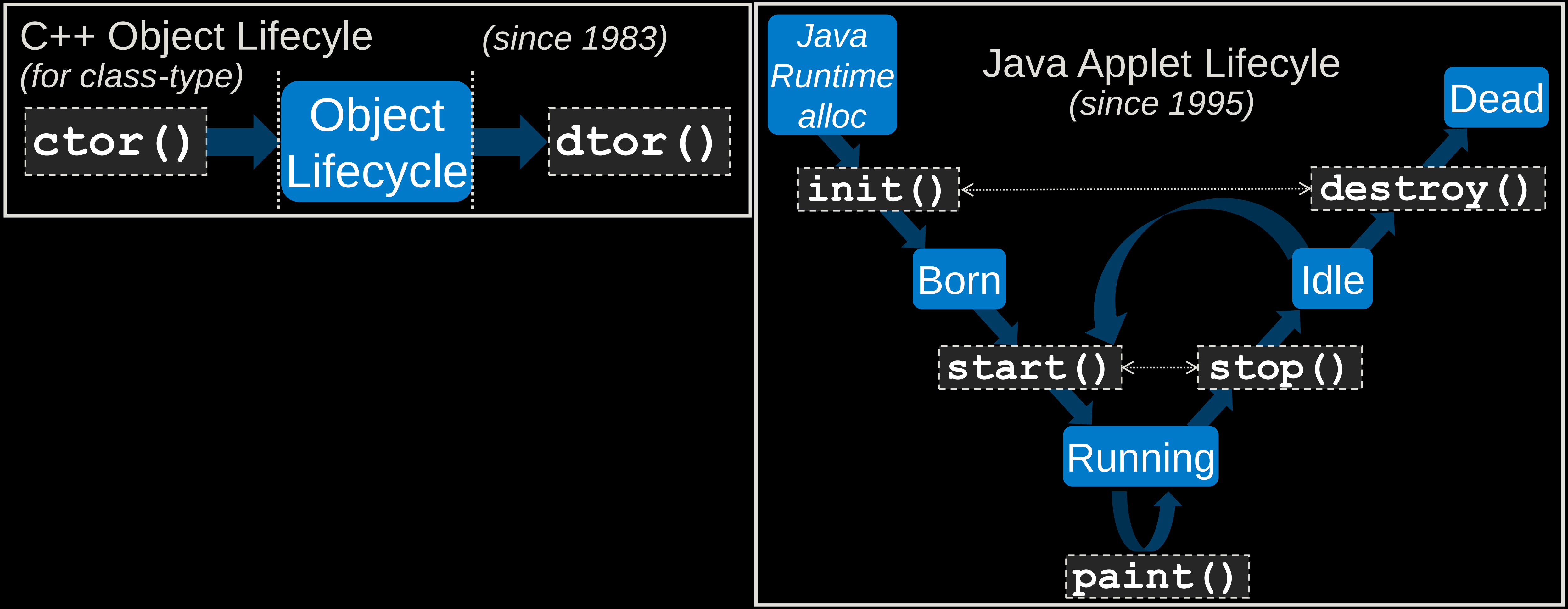
C++ Object Lifecyle *(since 1983)*
(for class-type)

`ctor()` → Object Lifecycle → `dtor()`

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

*…So Design or API is "**obvious**"*

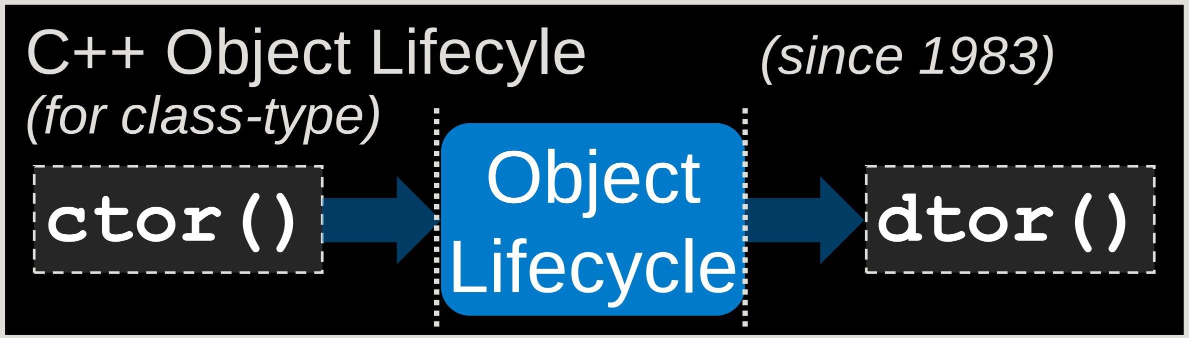
Expressing Symmetry Through API

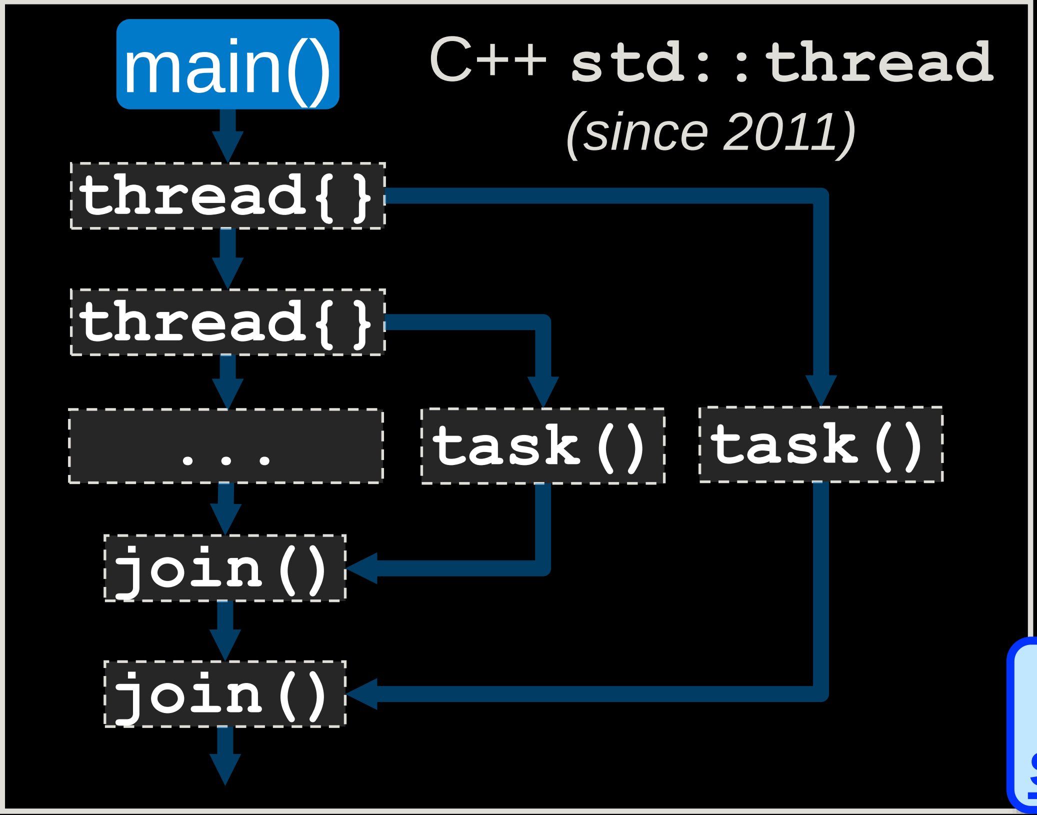


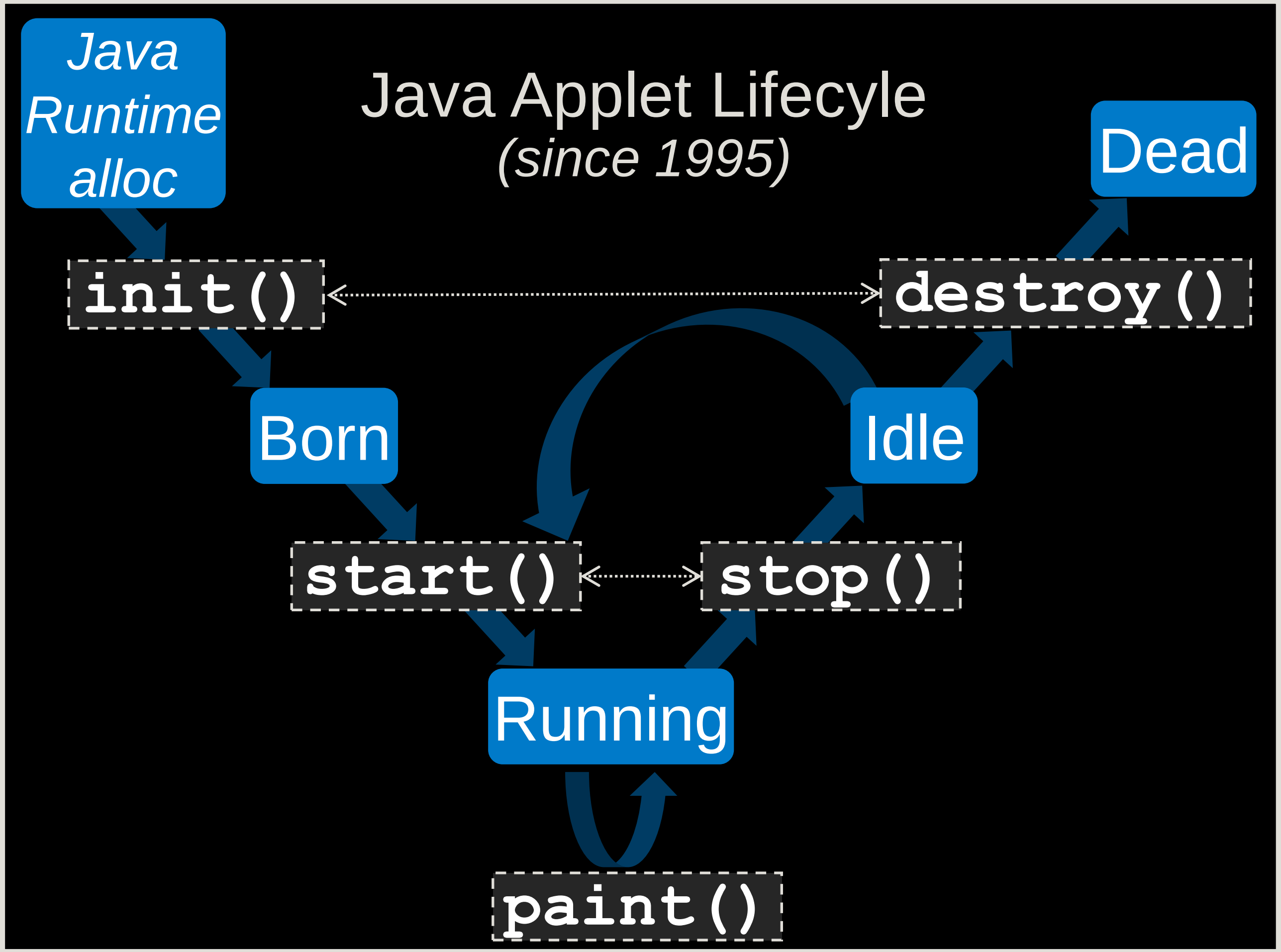
Goal: Find a way to emphasize **symmetry** in your Design and API

*…So Design or API is "**obvious**"*

Expressing Symmetry Through API







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

*...So Design or API is "**obvious**"*

Investigating Symmetry

`ctor()` ... `dtor()`

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

- 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

Decreasing Complexity

  - 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?

Increasing Complexity

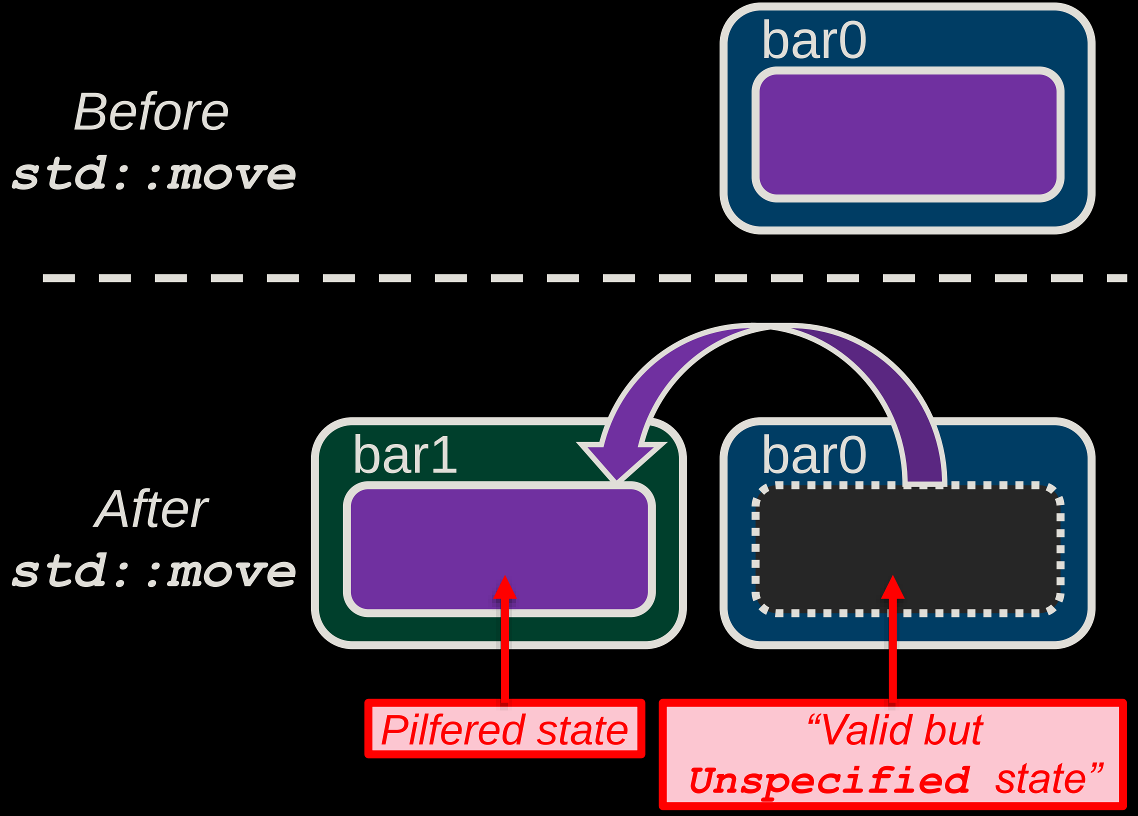
Role of Asymmetry

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);
```

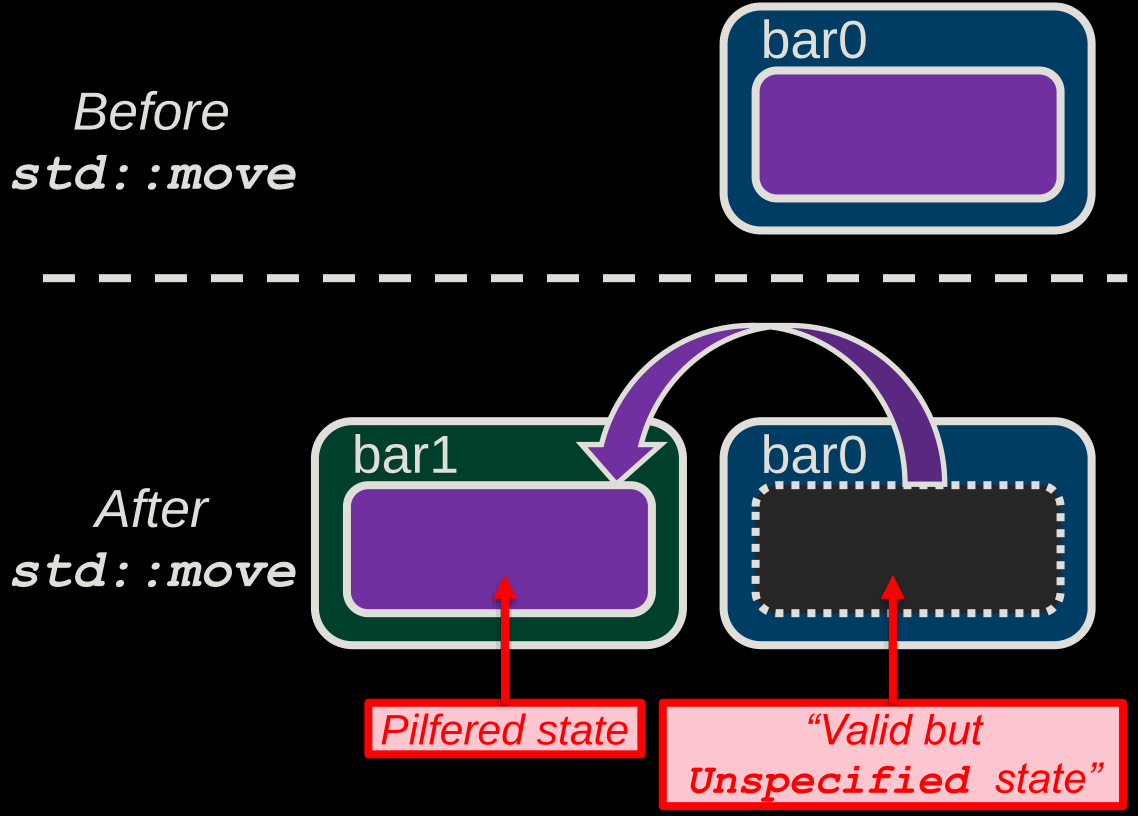


Cheating Symmetry: `std::move`

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);
```

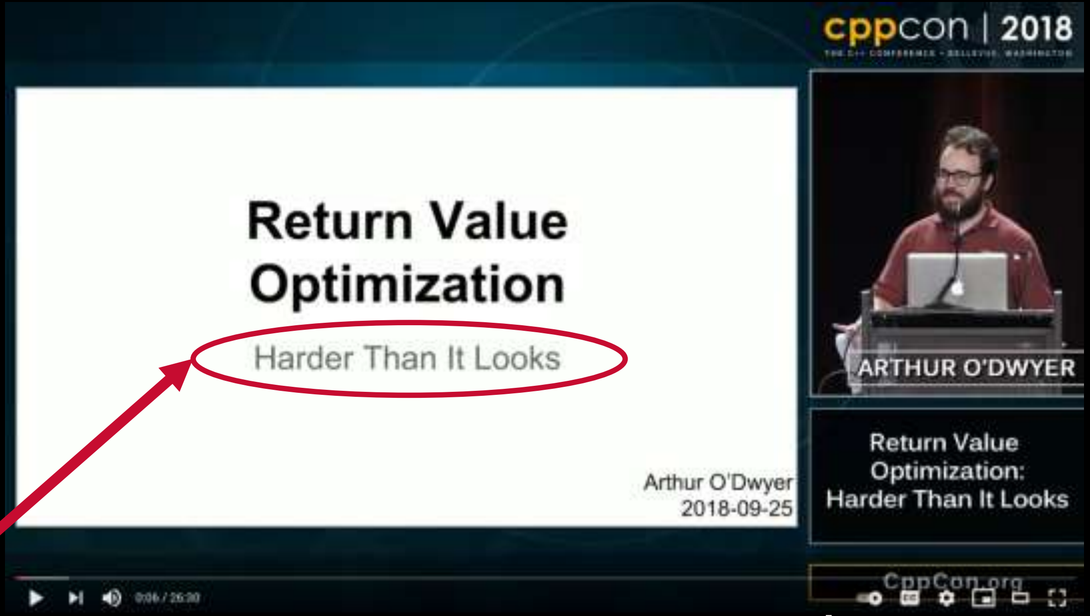
- **`std::move` is tricky** because:
  - Is **NOT symmetric** *(pilfered-from object has `Unspecified` mutation)*
  - Is **NOT orthogonal** *(pilfered-from and pilfered-into objects are related)*



`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)*



CppCon 2018: Arthur O'Dwyer
"Return Value Optimization: Harder Than It Looks"
https://www.youtube.com/watch?v=hA1WNtNyNbo

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

https://www.cppstories.com/2013/02/smart-pointers-gotchas/

The name for a symmetry violation:

Asymmetry

Asymmetry

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

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

Symmetric
- Harmonious or Balanced

Implies high predictability
(when pattern is recognized)

Asymmetric
- Unbalanced or Exceptional

Implies edge-cases or surprising behavior “at-scale”

Fiddler crab

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 intuiting**
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)

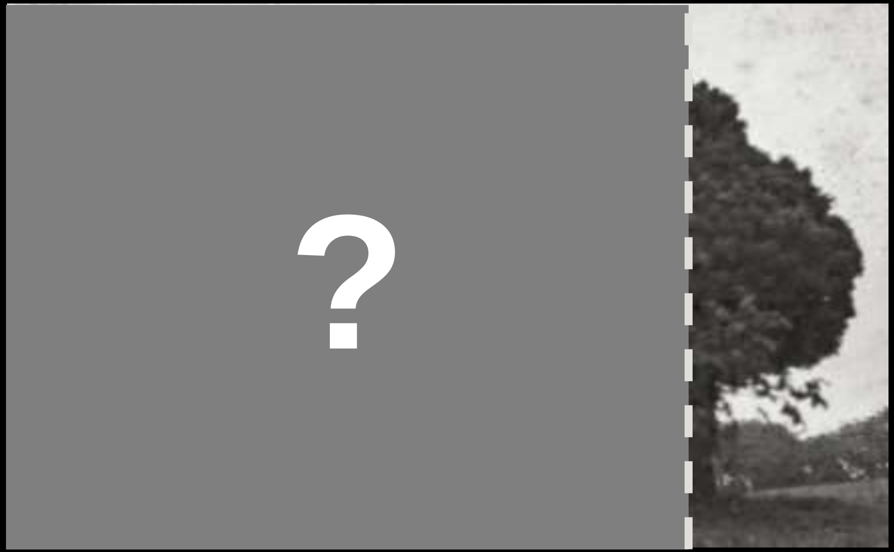
Asymmetry Examples

- Q: Guess what is hidden?


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

Asymmetry Examples

- Q: Guess what is hidden?



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

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

Asymmetry Examples

- Q: Guess what is hidden?

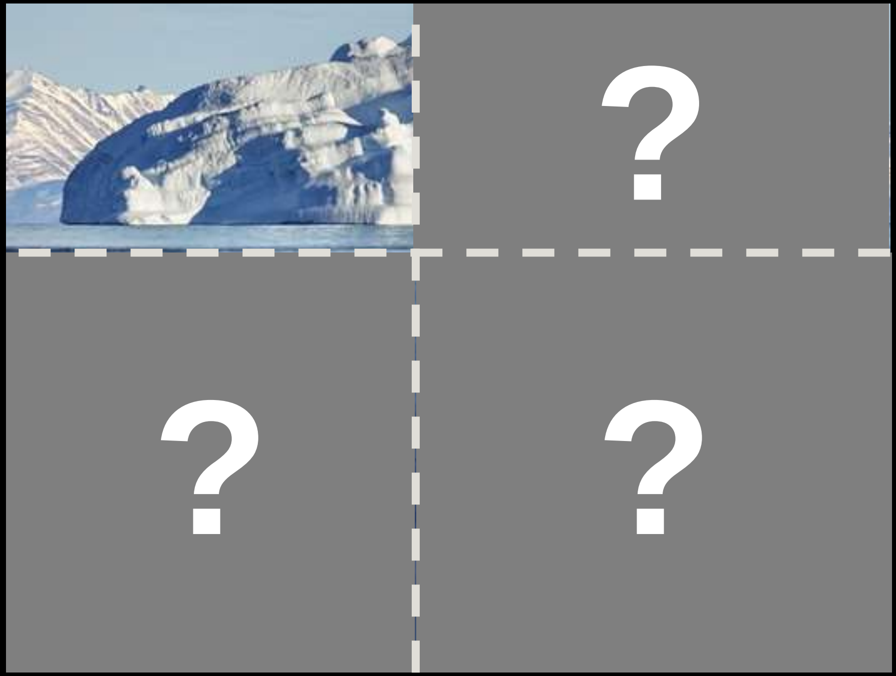
https://www.flickr.com/photos/128139955@N02/50811516651

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

Asymmetry Examples

- Q: Guess what is hidden?

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

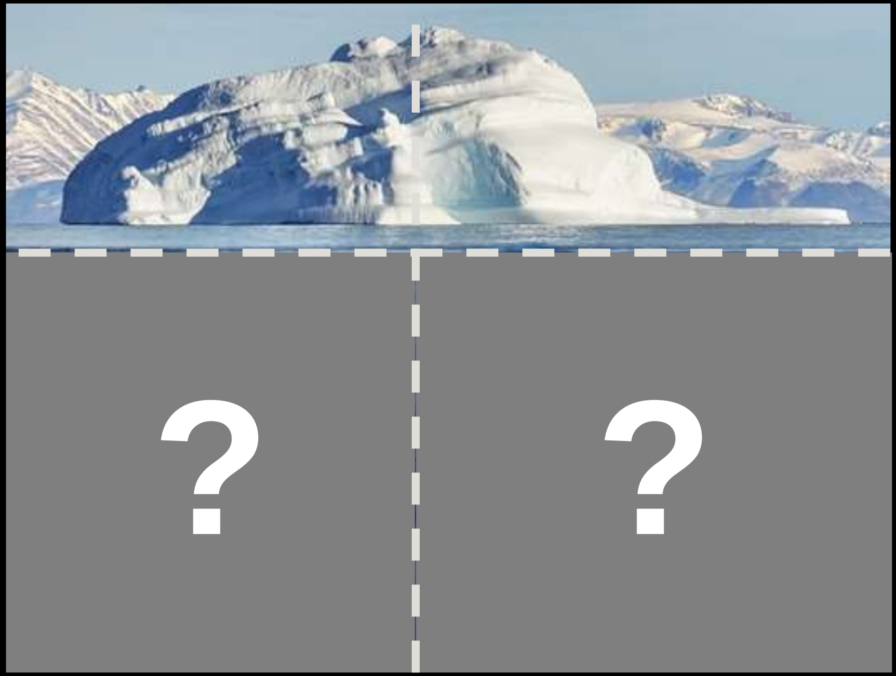


Asymmetry **prevents intuiting**
about what we do not see
(based on what we see)

Asymmetry Examples

- Q: Guess what is hidden?

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

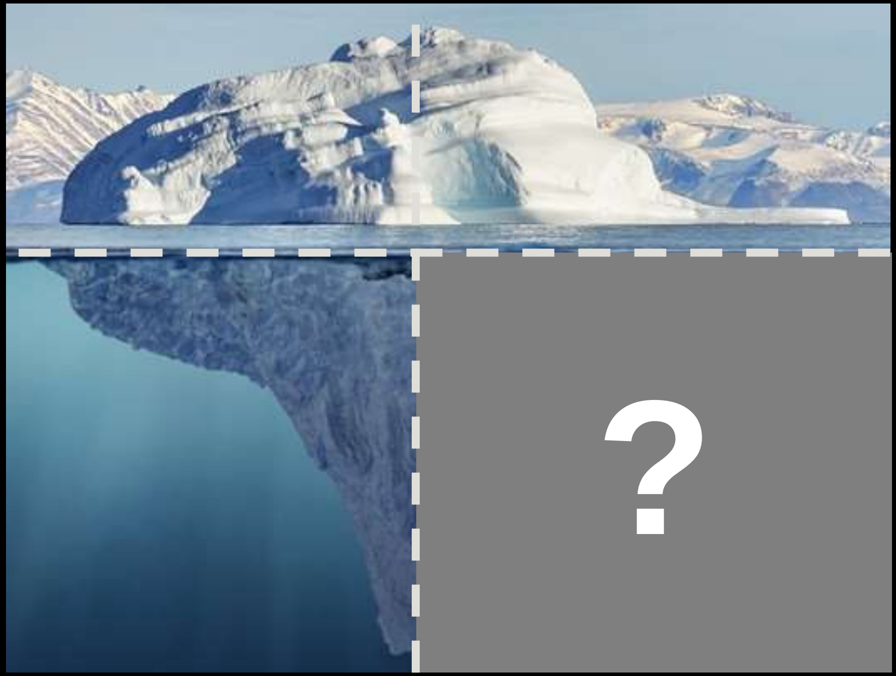


Asymmetry **prevents intuiting**
about what we do not see
(based on what we see)

Asymmetry Examples

- Q: Guess what is hidden?

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



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

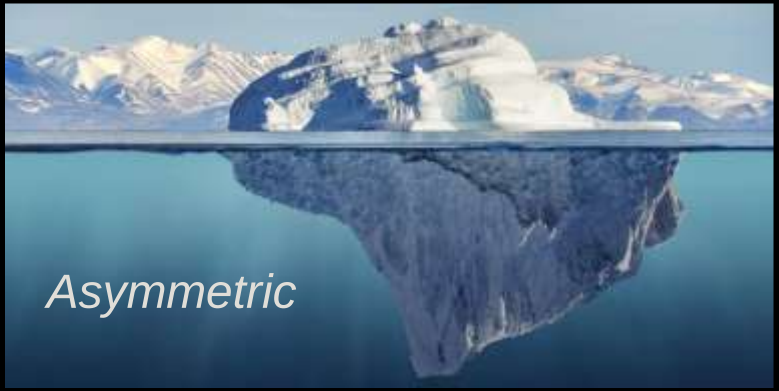
Asymmetry Examples

- Q: Guess what is hidden?

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

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 <u>edge cases</u> and <u>surprising behavior</u>

Examples of **<u>asymmetry in C++</u>**:

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

<u>`std::string_view`</u> *vs.* <u>`std::span<>`</u>

- 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: <u>http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1085r2.md</u>)*

Globals are inherently ***<u>asymmetric</u>***

<u>C++ Globals</u>

- Construction order:
  - Thread-safe for static instance at block scope *(since C++11)*
    - Reentrancy invokes undefined behavior, *see:* <u>https://devblogs.microsoft.com/oldnewthing/20040308-00/?p=40363</u>
  - 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

<u>Hygiene</u> *(def):* Conditions and practices to promote health

- Asymmetry examples in ***<u>Your Codebase</u>***:

1 <u>Design</u> Hygiene

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

2 <u>Implementation</u> Hygiene

- **<u>Unnecessarily complex processing</u>**
  - Example: Eager-compute or Lazy-compute that introduces stochastic time-shifting of computation and resource contention, or which encourages branch mis-prediction
- **<u>Multiple function returns</u>** *(resulting in different control flows within function)*
- **<u>Multiple abstraction levels within a given function body</u>**

"Every statement in a function body should be at the same level of abstraction." (paraphrased)

Tony Van Eerd
@tvaneerd

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)*

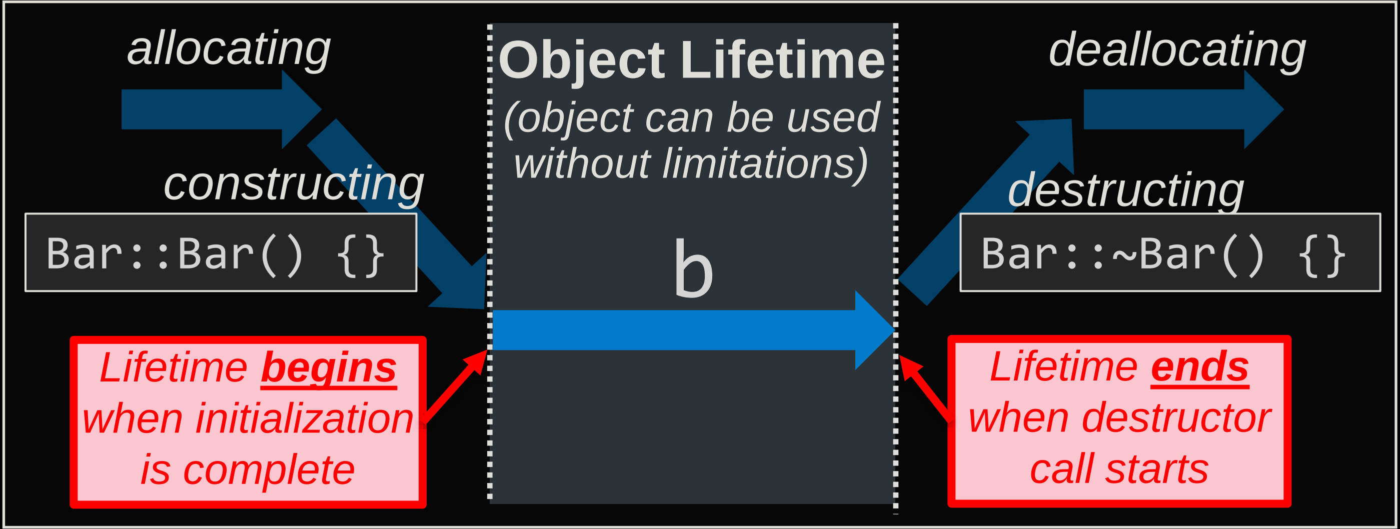
Register storage duration is automatic plus compiler hints *(until C++17)*

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;
```



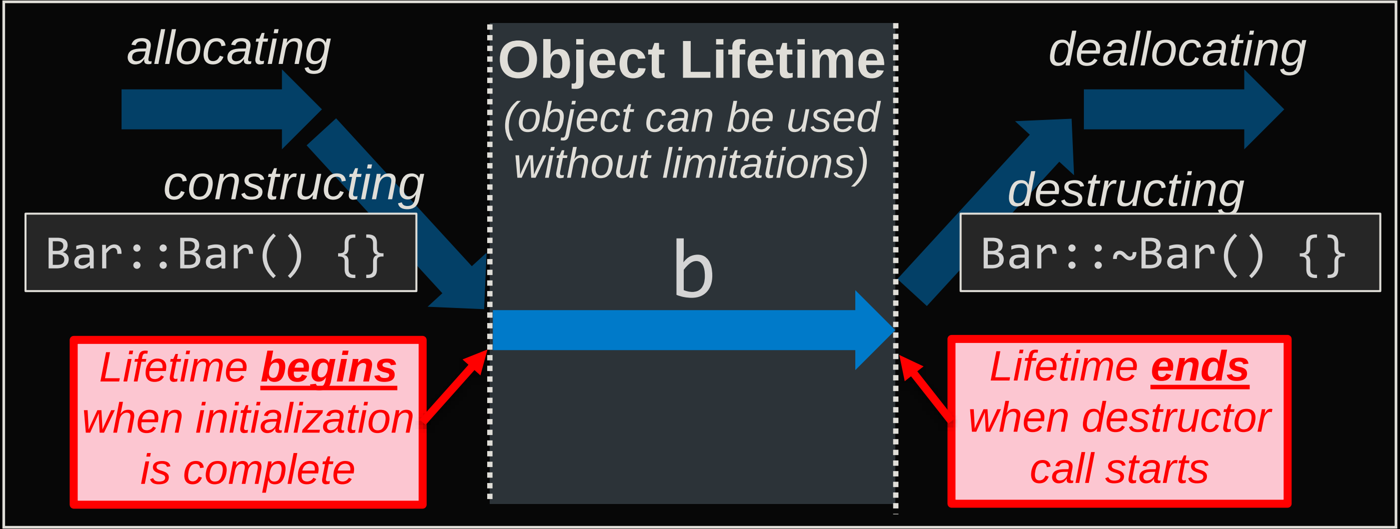
Review: Storage Duration

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

- **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;
```



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)*

Object Lifetime

Begins when **BOTH** of:

1. Storage is obtained
2. Initialization is complete

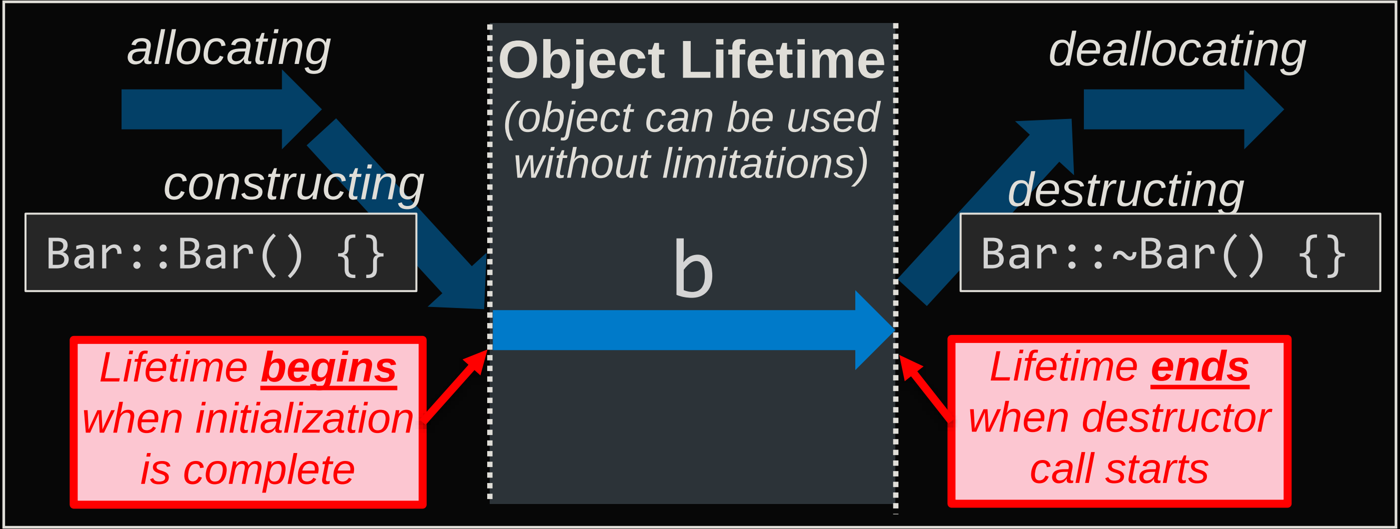
Ends when **EITHER** of:

1. Dtor starts
2. Storage is reused or released

*Is actually **symmetric** (what you would expect to enforce type invariants)*

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

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



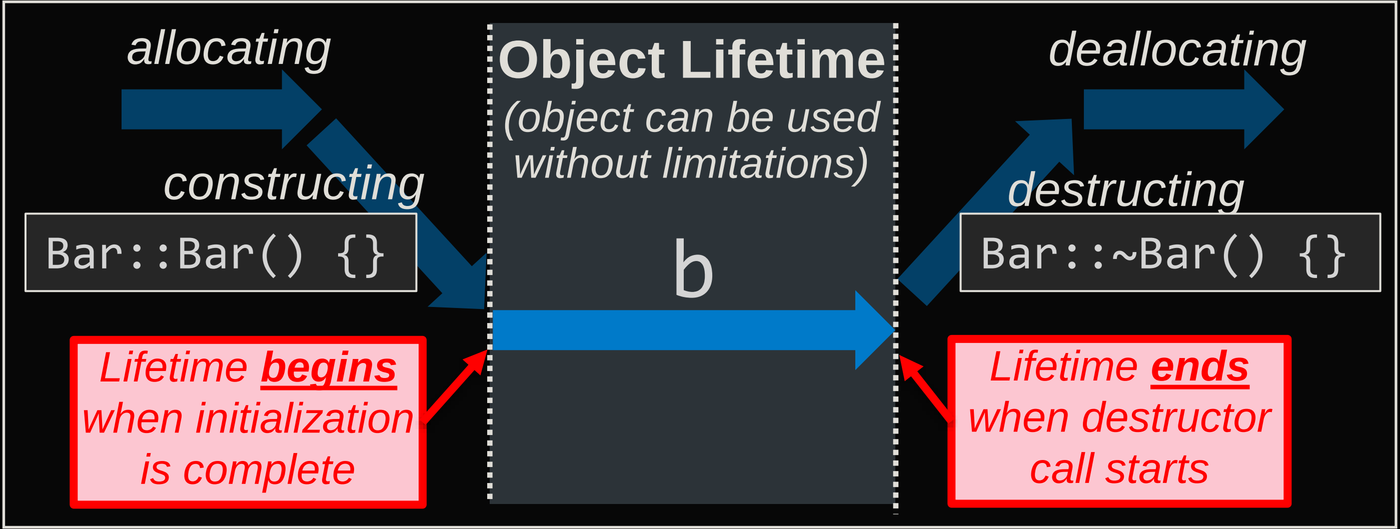
Review: Storage Duration

- **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;
```

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



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

*Is actually **symmetric** (what you would expect to enforce type invariants)*

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)*

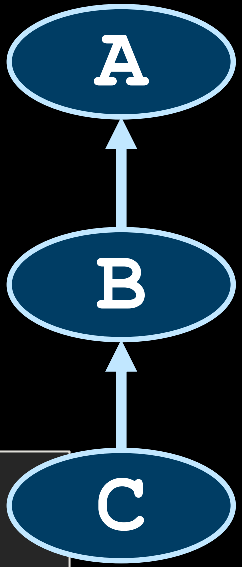
Hierarchy Hygiene

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

```
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!)

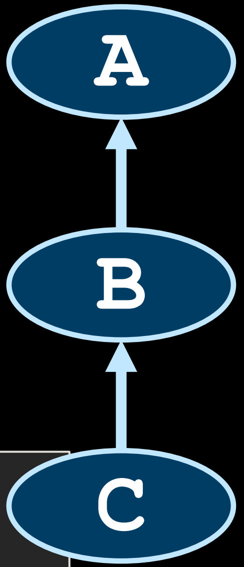
Special **edge cases** exist for hierarchy **implementation**

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

```
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!)

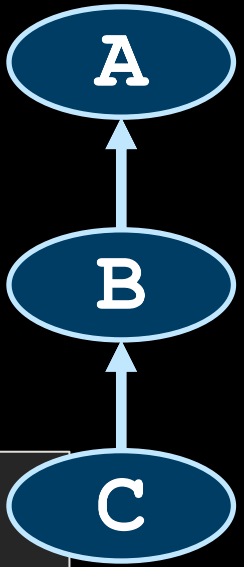
Special **edge cases** exist for hierarchy **implementation**

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

```
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"***

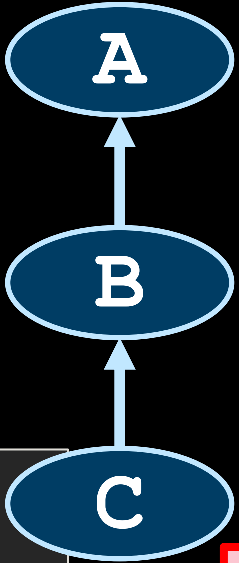
Hierarchy Hygiene

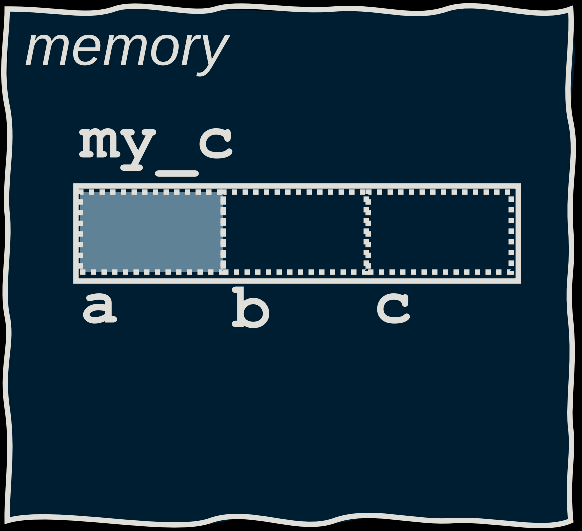
- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

```
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

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

allocating

constructing

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

***A** exists*

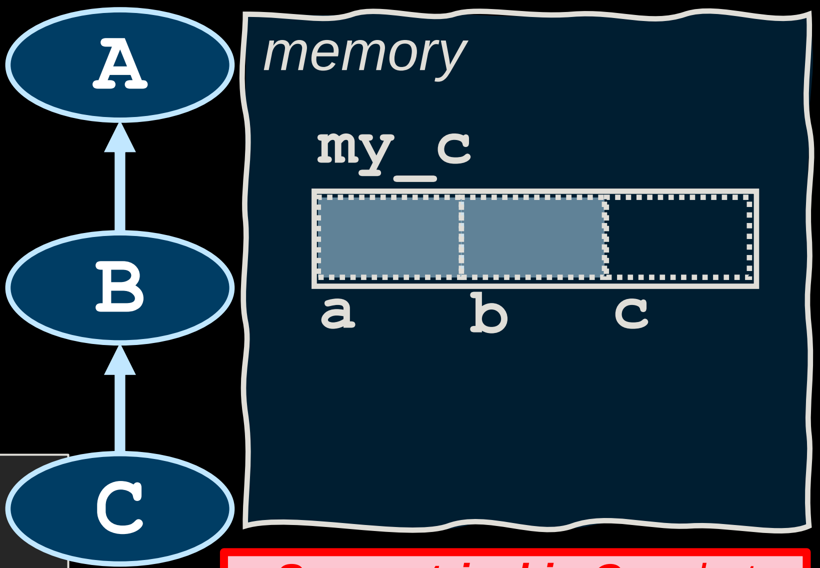
Hierarchy Hygiene

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

```
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***

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

allocating

constructing

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

A exists

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

B exists

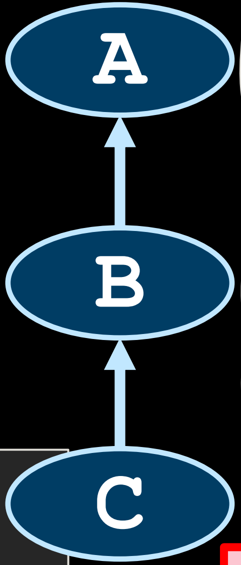
Hierarchy Hygiene

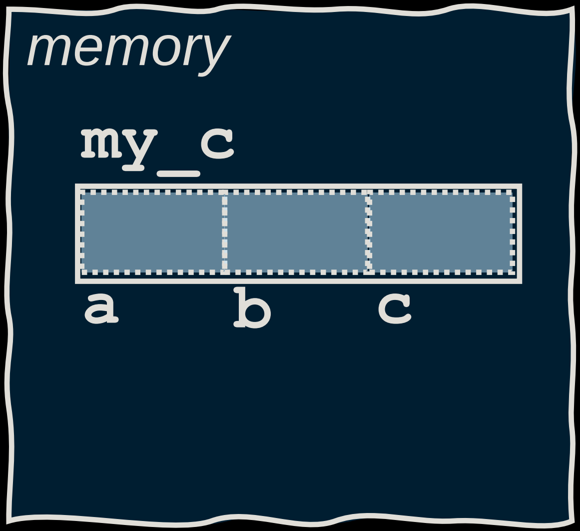
- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

```
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

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

allocating

constructing

Object Lifetime
(object can be used without limitations)

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

***A** exists*

***B** exists*

***C** exists*

*Lifetime **begins** when initialization is complete*

my_c

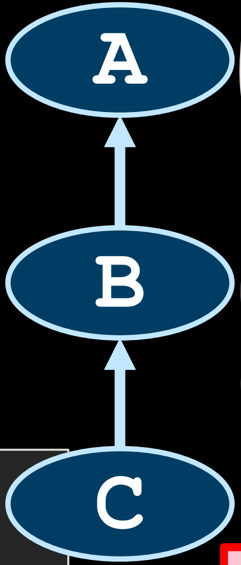
Hierarchy Hygiene

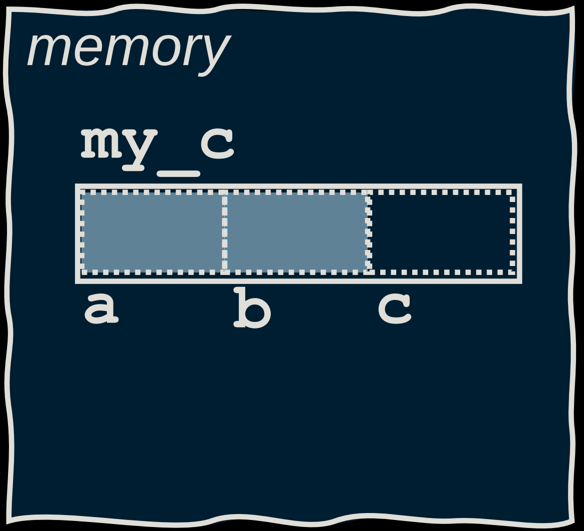
- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

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

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

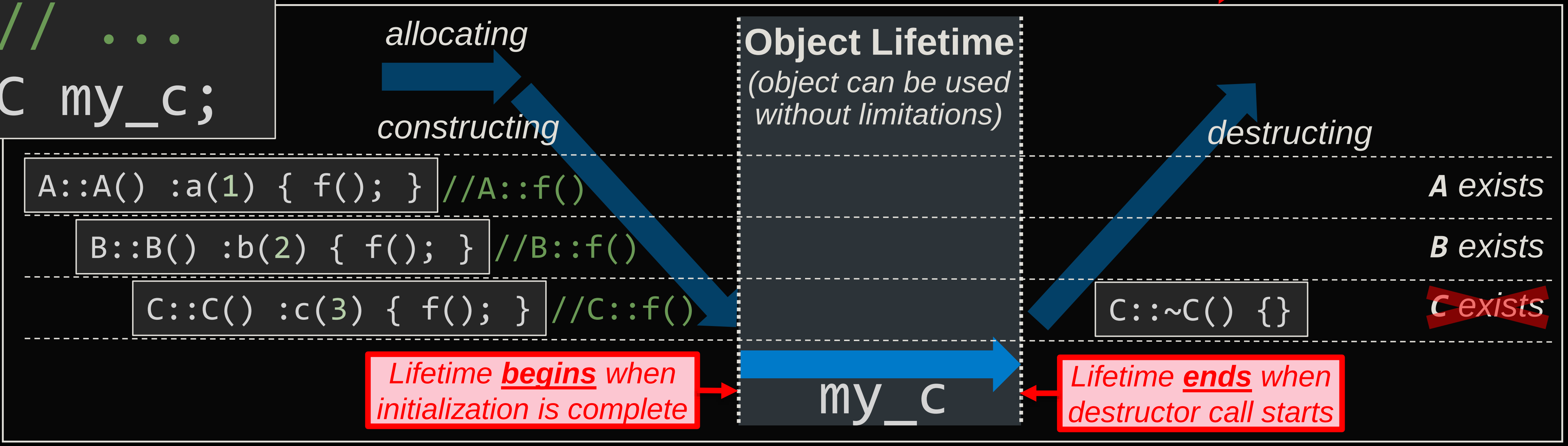
```cpp
class C : public B {
  public:
    void f() override;
    int c;
};
```





__Symmetrical in C++__, but Your Implementation may have assumed __asymmetrical__

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



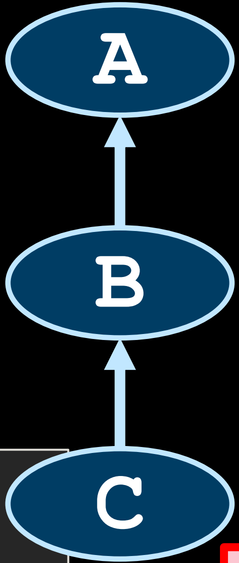
Hierarchy Hygiene

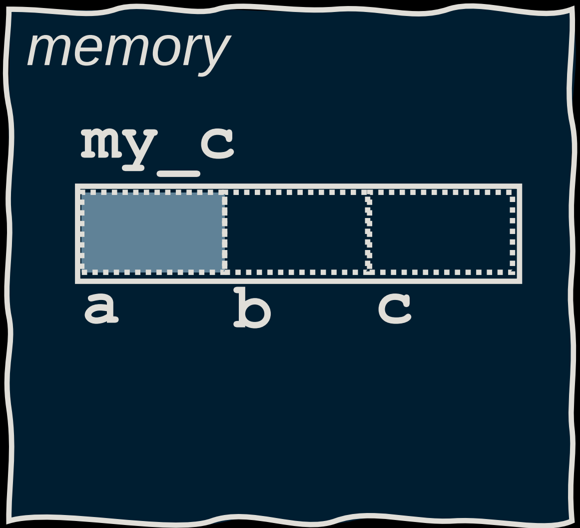
- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

```
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__

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

allocating

constructing

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

Object Lifetime
(object can be used without limitations)

my_c

destructing

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

A *exists*

~~***B*** *exists*~~

~~***C*** *exists*~~

Lifetime __begins__ when initialization is complete

Lifetime __ends__ when destructor call starts

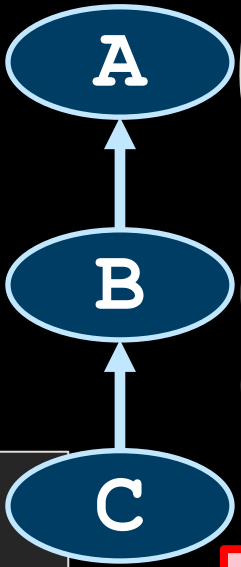
Hierarchy Hygiene

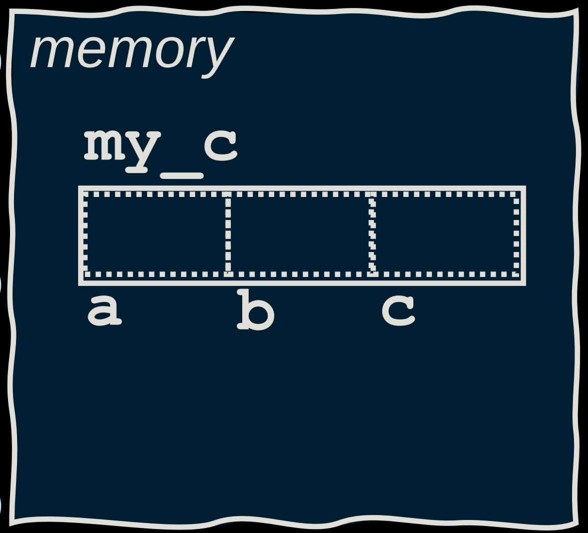
- **C++ inheritance** exhibits **outstanding symmetry** *(Guaranteed!)*
- Special **edge cases** exist for hierarchy **implementation**

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

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

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





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

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

allocating

constructing

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

Object Lifetime
(object can be used without limitations)

my_c

*Lifetime **begins** when initialization is complete*

deallocating

destructing

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

~~A exists~~
~~B exists~~
~~C exists~~

*Lifetime **ends** when destructor call starts*

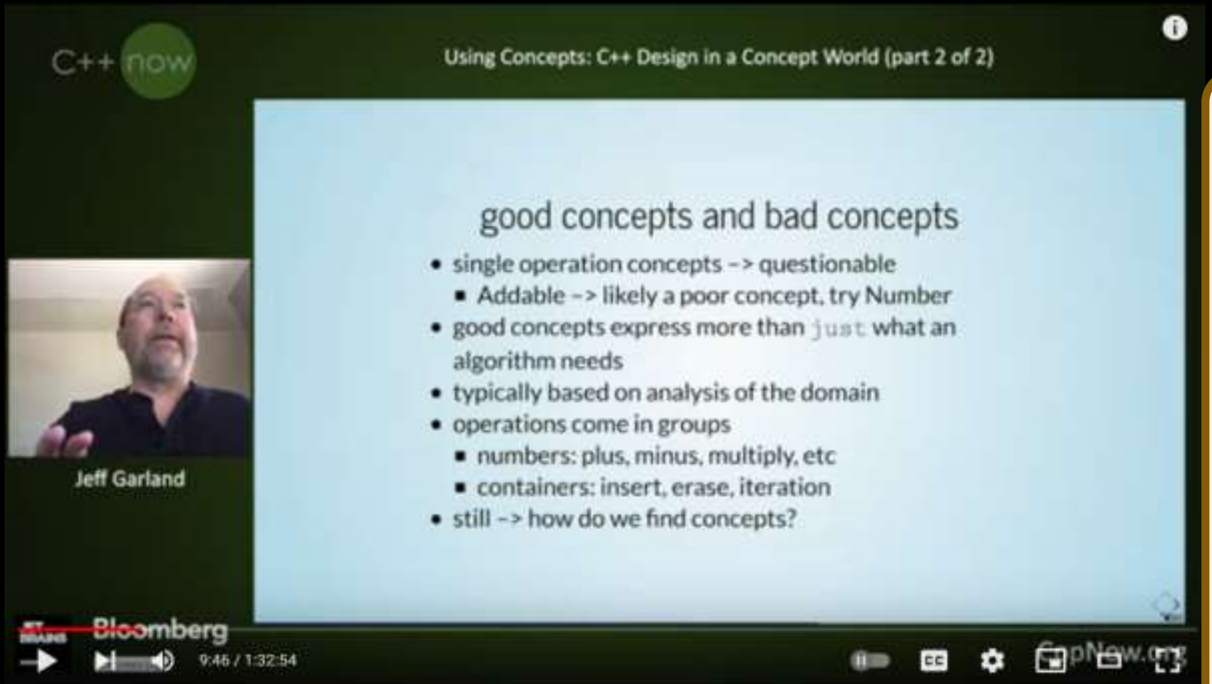
Other Asymmetry in Class Hierarchies

- 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`**?



Using Concepts: C++ Design in a Concept World (part 2 of 2) - Jeff Garland - [CppNow 2021]

https://www.youtube.com/watch?v=IXbf5lxGtr0

> *"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**

Dutch mathematician and computer scientist:

- *Numerical analysis*
- *Programming languages*
- *Design principles*

Adriaan "Aad" van Wijngaarden (1916-1987)

> *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.*

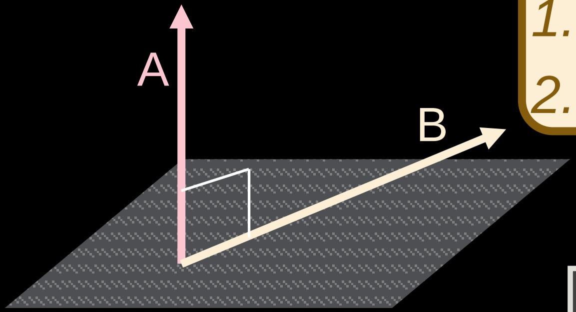
-- Adriaan van Wijngaarden et al., Revised Report on the Algorithmic Language ALGOL 68, section 0.1.2, Orthogonal design

Want to Learn More?

See: Edsger W. Dijkstra ***Separation of Concerns***

Orthogonality In Practice

Orthogonal:
- Unrelated *(no relation exists)*



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

At right-angles

- 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)

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

Make things "unrelated"

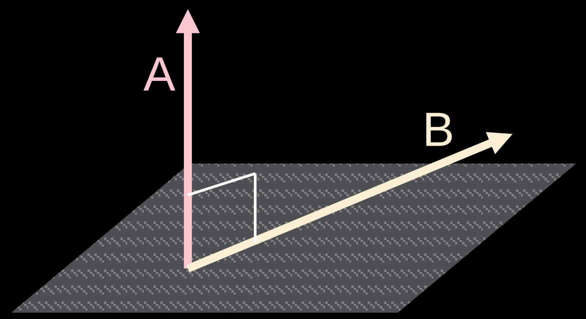
Orthogonality grants simplicity to ***dismiss as a possibility*** *some behaviors or component interactions within the resulting system.*

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

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

Orthogonality allows us to **dismiss as a possibility** some behaviors or component interactions

(without tedious inspection)

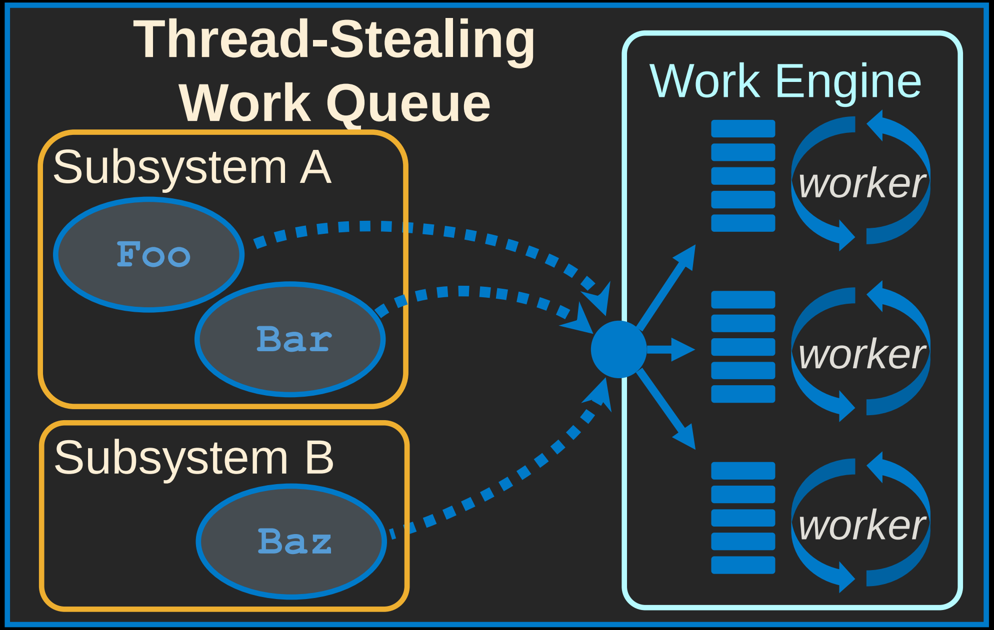


Case Study: Thread-Stealing Work Queue

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)*



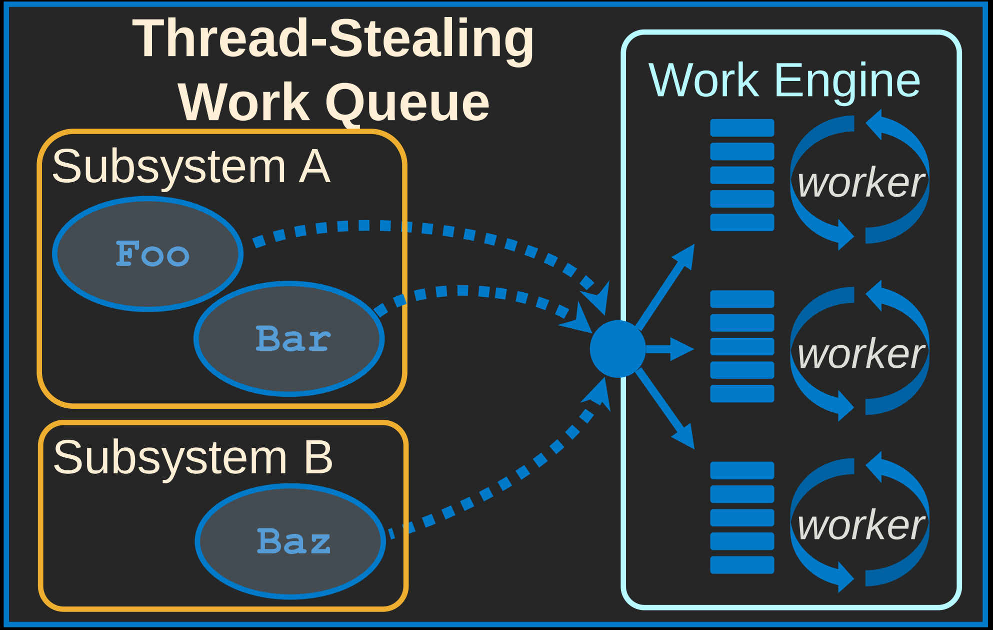
Is great design because is both symmetrical and orthogonal

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

Symmetry:

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

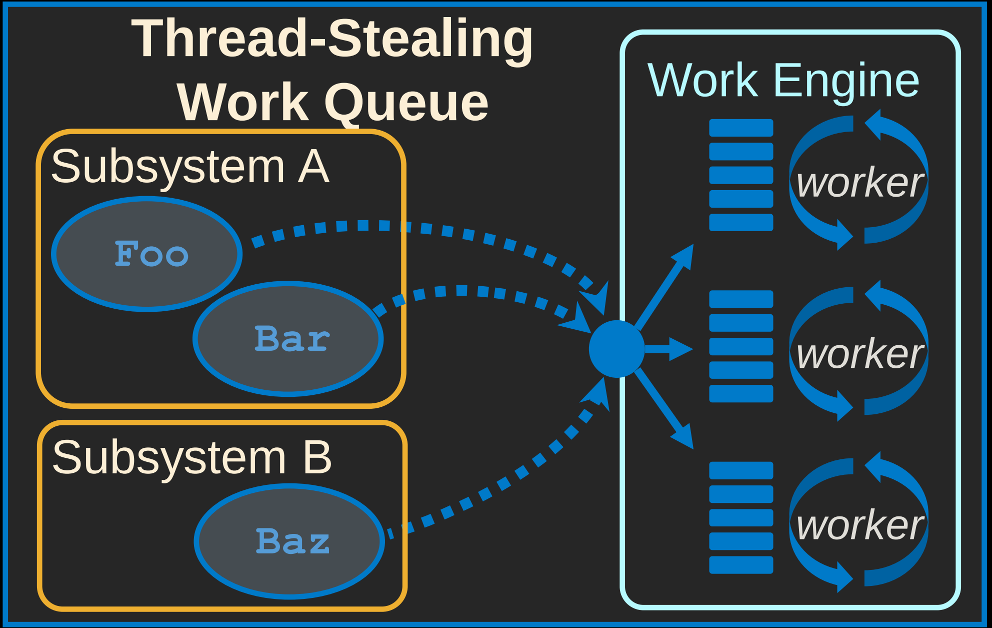
- **Benefits**: Creating work is obviously correlated with understanding for how work is completed *(consumed)*
- **Costs**: Special handling is required to violate design symmetry when asymmetry is desired *(such as special handling to execute one work item many times)*

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

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)*

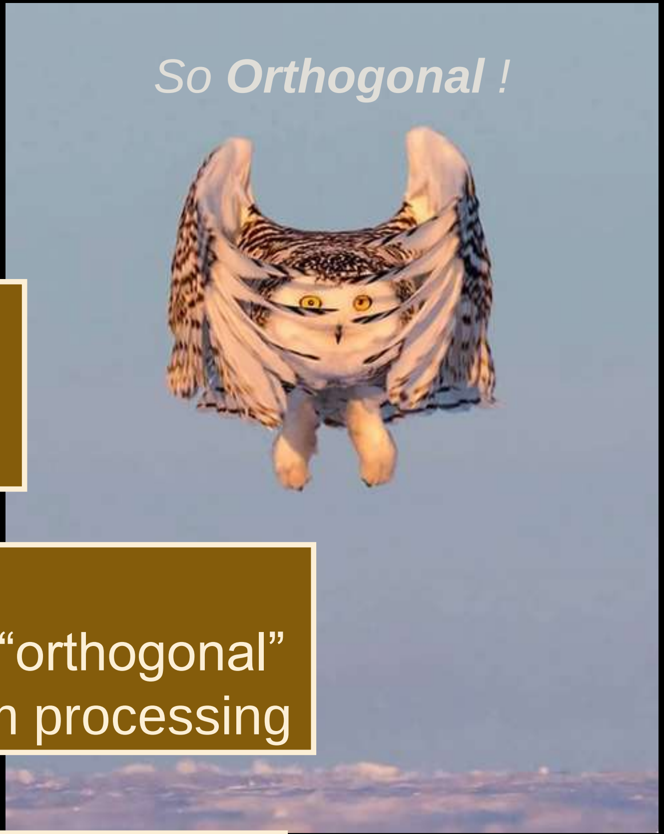
Orthogonality:

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

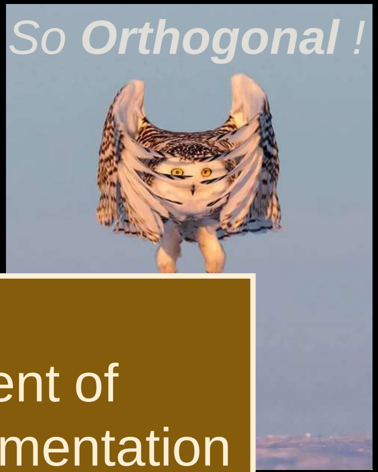
Trigger-interface APIs:
Trigger, handler, or callback invokes subsystem orthogonally to normal system execution flow *(e.g., handling raised exceptions, system events, queued callbacks)*

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

PREFERED **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

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

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

Design Relationships

All ***Possible*** *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 **A**, I know **B** — **Strong** Relationship *(is guaranteed)*

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

Useful *and* ***Robust!***

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)*

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)*

Tricky!

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*

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

Knowing "something" can be more dangerous than knowing "nothing"
(orthogonality provides stronger guarantees)

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)*

Tricky!

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*

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!***

Prefer Stronger Two-Way Relationships

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

Why?
- *Greater "Knowing"*
- *Fewer assumptions*
- *Reduced edge cases*

Bidirectional Symmetry

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

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

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

- **<u>Prefer Stronger Guarantees</u>** *(one of):*
  - *(strong-)*Symmetry
  - Orthogonality

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

Bidirectional Symmetry

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

Relationships

In Theory…

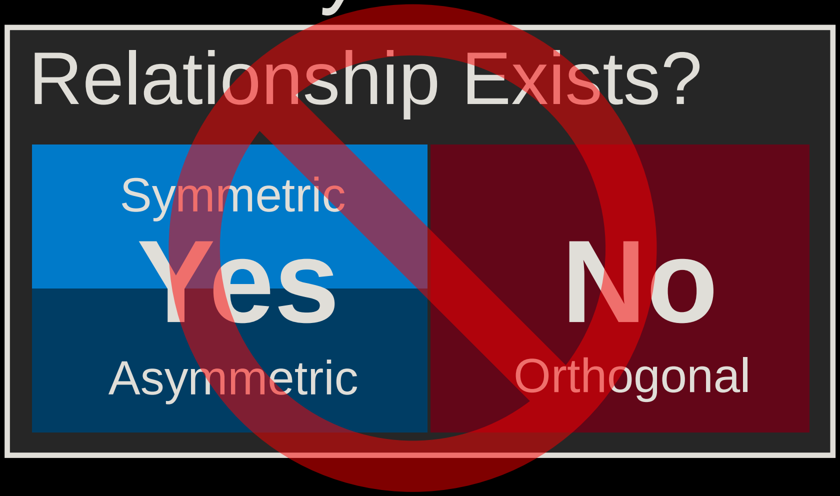
Relationships can be **complicated**

Relationship Exists?

Yes	No
Symmetric	Orthogonal
Asymmetric	

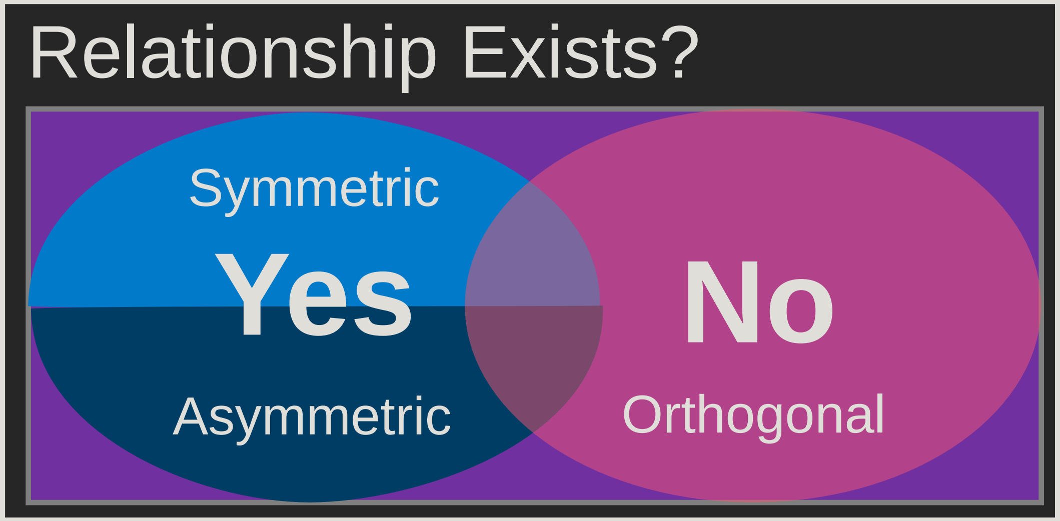
Relationships

In Theory…



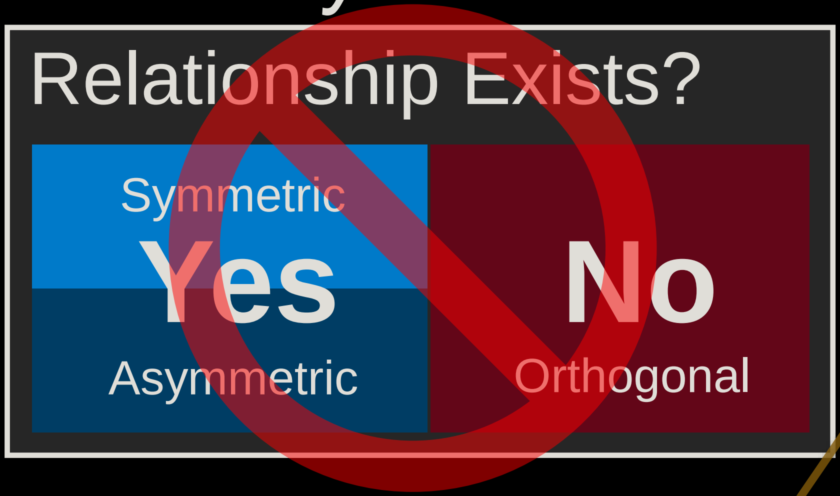
Relationships
can be **complicated**

In Reality…



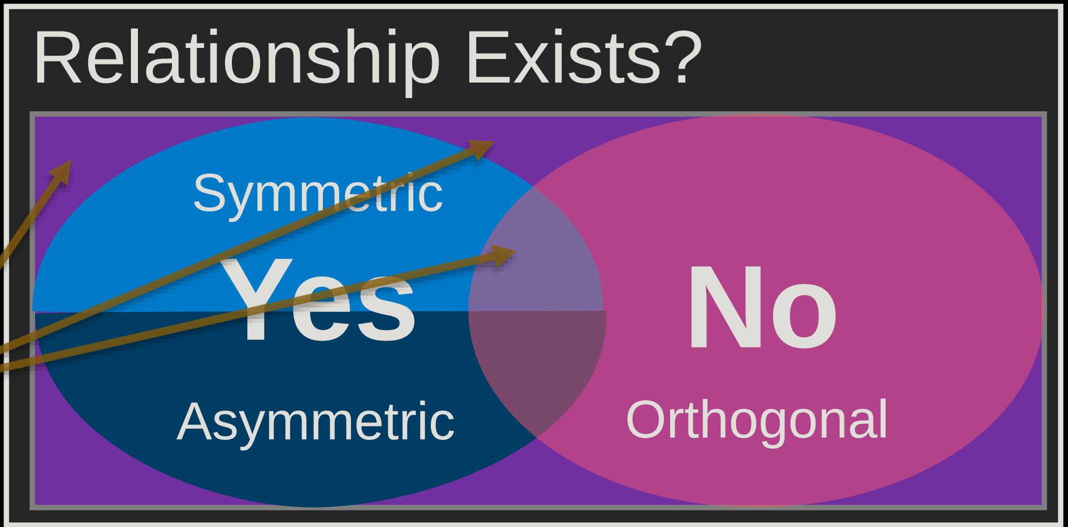
Relationships

In Theory…



Relationships
can be **complicated**

In Reality…



Not perfectly Symmetric

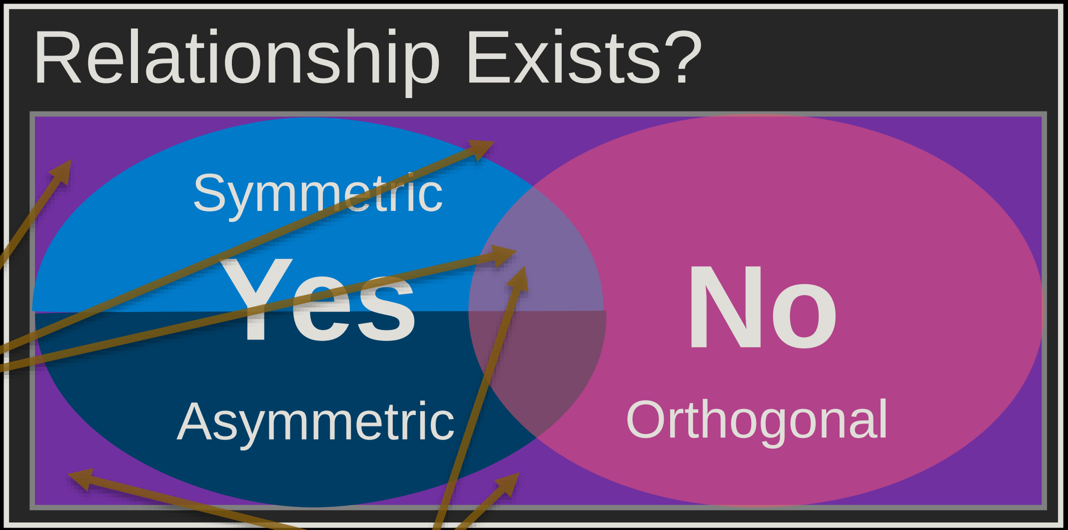
- **Create…destroy** *(e.g., globals)*
- **Some operations do not fully go backwards**
  - Init…uninit
  - Load…unload
  - Start…shapshot-save…shutdown
- **Lifecycle Exception or Error Tracing**
  - Source…sink
  - Producer…consumer
  - Push…pull *(how to leak back failed operation from previous push)*

Relationships

In Theory…

Relationships can be **complicated**

In Reality…



Not perfectly Symmetric

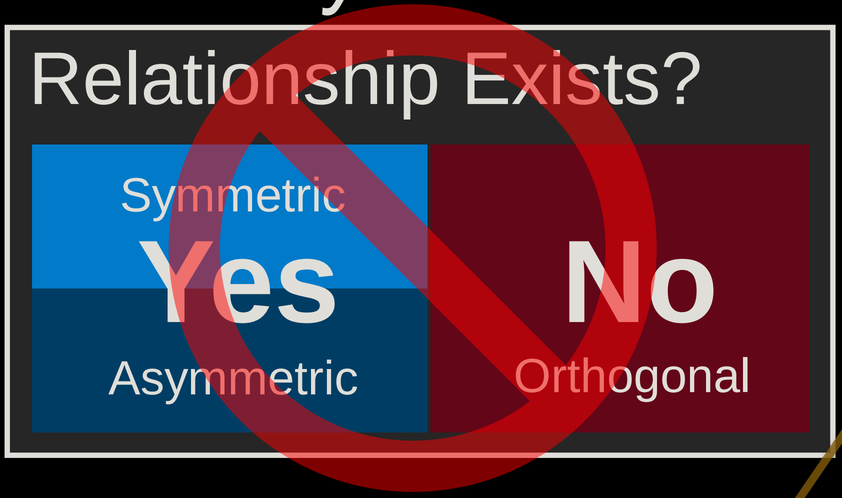
- **Create…destroy** *(e.g., globals)*
- **Some operations do not fully go backwards**
  - Init…uninit
  - Load…unload
  - Start…shapshot-save…shutdown
- **Lifecycle Exception or Error Tracing**
  - Source…sink
  - Producer…consumer
  - Push…pull *(how to leak back failed operation from previous push)*

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

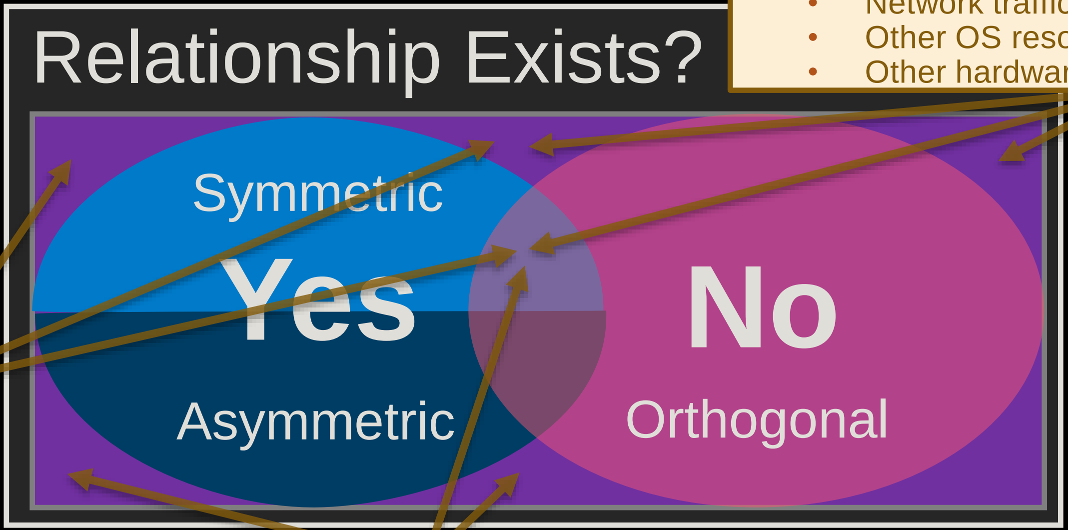
Relationships

In Theory…



Relationships
can be **complicated**

In Reality…



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

Not perfectly Symmetric

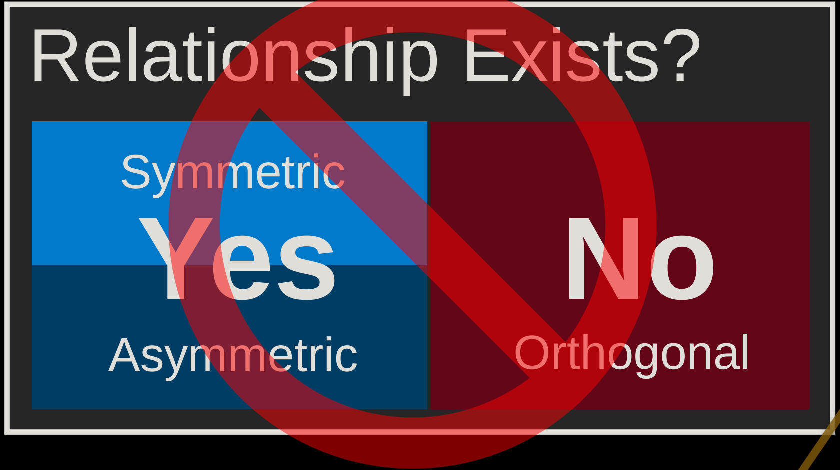
- **Create…destroy** *(e.g., globals)*
- **Some operations do not fully go backwards**
  - Init…uninit
  - Load…unload
  - Start…shapshot-save…shutdown
- **Lifecycle Exception or Error Tracing**
  - Source…sink
  - Producer…consumer
  - Push…pull *(how to leak back failed operation from previous push)*

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

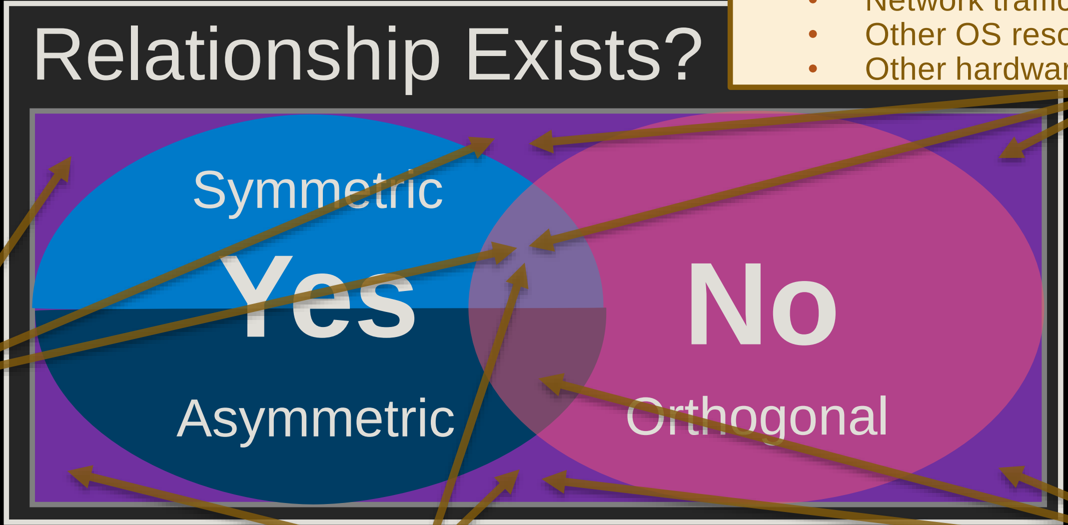
Relationships

In Theory…



Relationships
can be **complicated**

In Reality…



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

Not perfectly Symmetric

- **Create…destroy** *(e.g., globals)*
- **Some operations do not fully go backwards**
  - Init…uninit
  - Load…unload
  - Start…shapshot-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 asymmetric correlations

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

Relationship Space

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

***(Huge!)* Design Space**
to define relationships
(one-way, two-way)

Relationship Space
(one way: A ➔B)

Relationship Coupling

None Weak Strong

Relationship Symmetry

Relationship Space

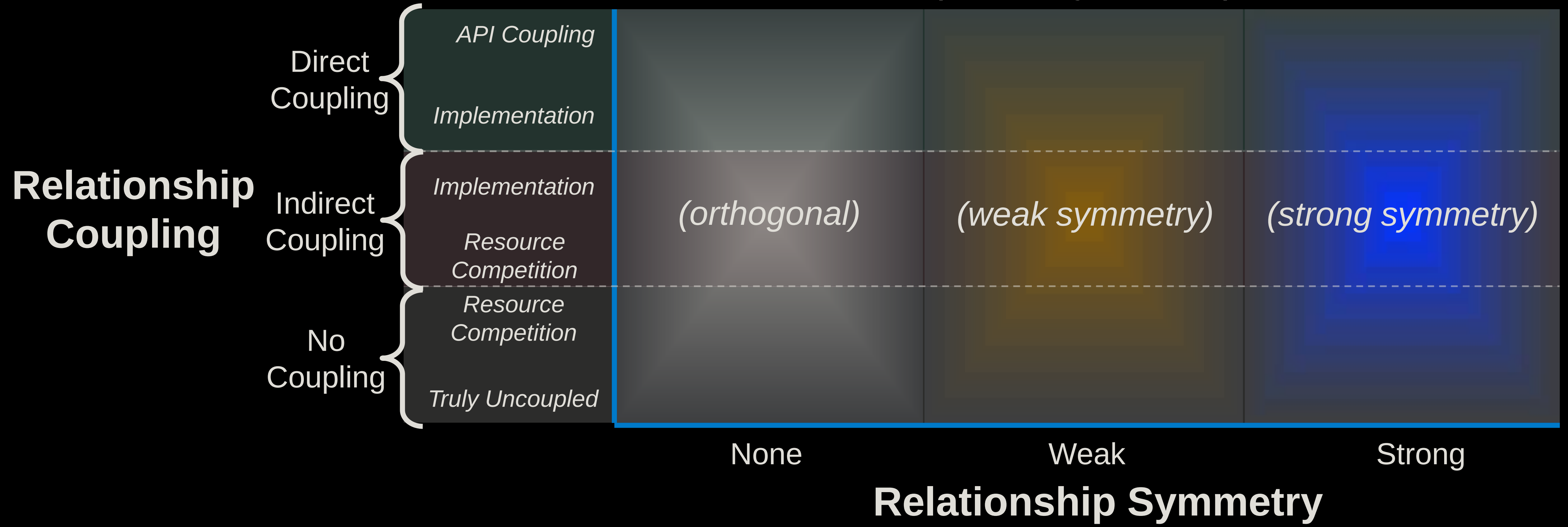
Relationship Attributes:

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

(Huge!) Design Space
to define relationships
(one-way, two-way)

Relationship Space

(one way: A ➔B)



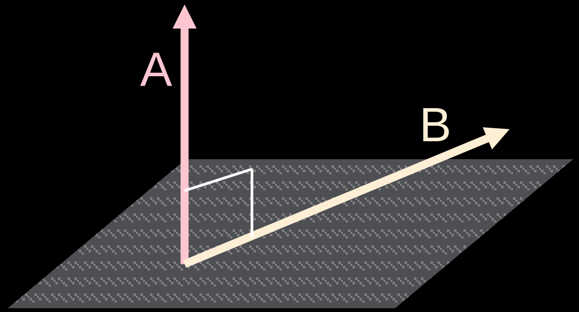
Conclusion

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

YES: *Intuitive*

NO: *Tedious inspection required*

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`")*

Leveraging Symmetry And Orthogonality

- **Symmetric**: Balanced relationship
- **Asymmetric**: Unbalanced relationship
- **Orthogonal**: No relationship

- 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

Design: ***"How It Works"***

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:

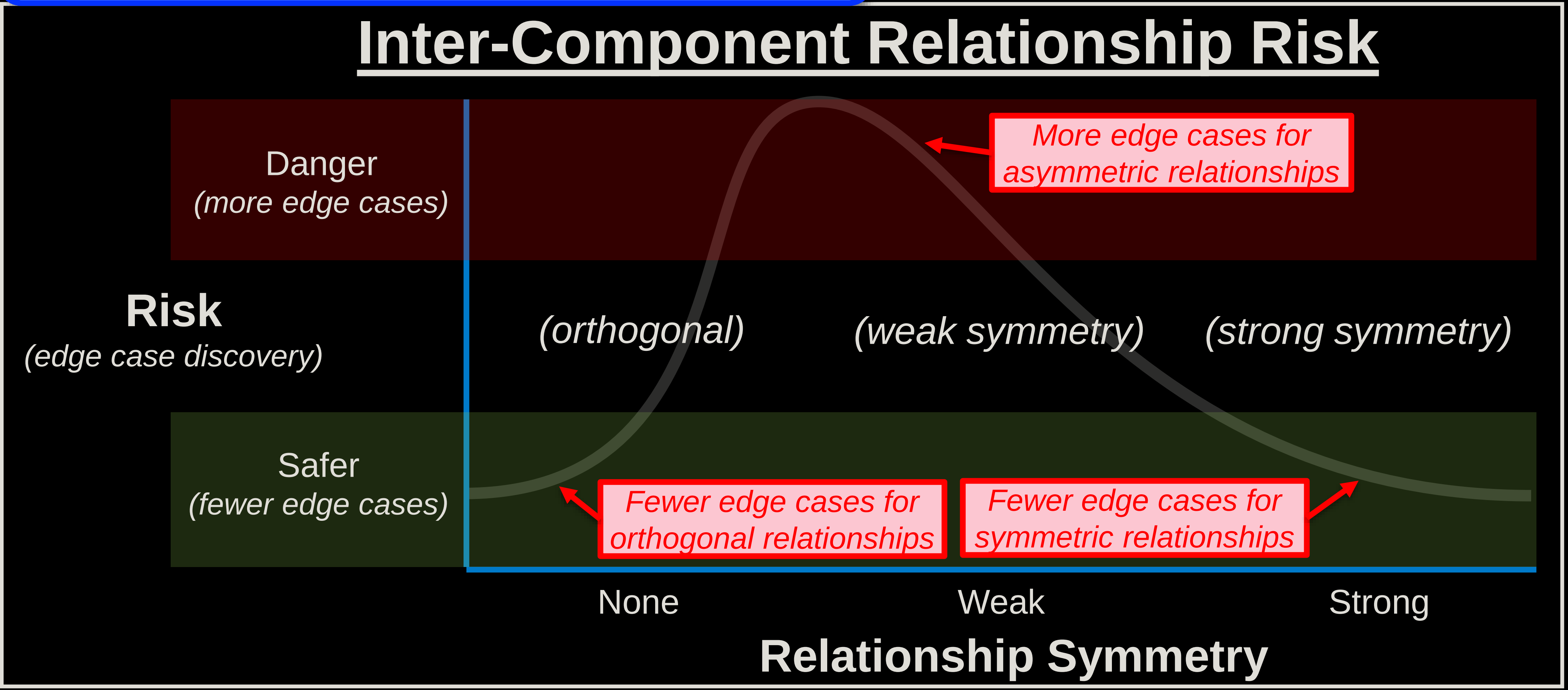
- Entirely independent: Network traffic flows and frame-draw
- Entirely independent: Log traffic processing and main thread
- Entirely independent: Work item processing and allocator amortization

Reducing Risk

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

Inter-component edge cases tend to present when system is:

- At-Scale
- Under-Load



Design In-Practice

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

1. **State** relationship
2. **Control-flow** relationship

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

Design In-Practice

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

1. **State** relationship
2. **Control-flow** relationship

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**

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?