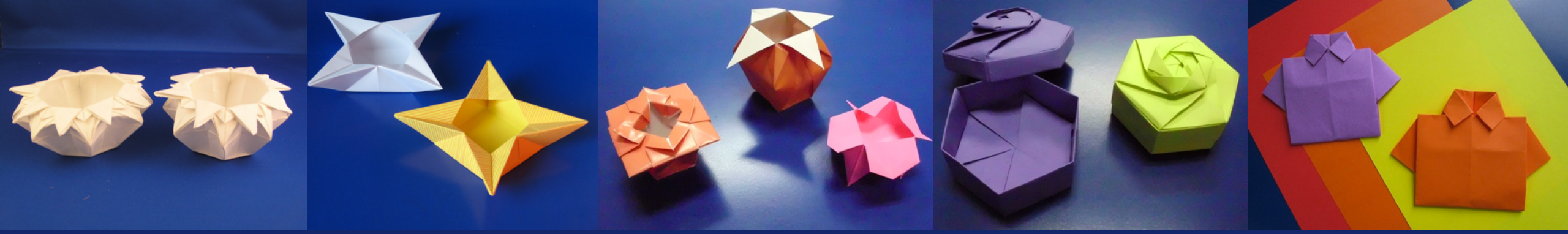
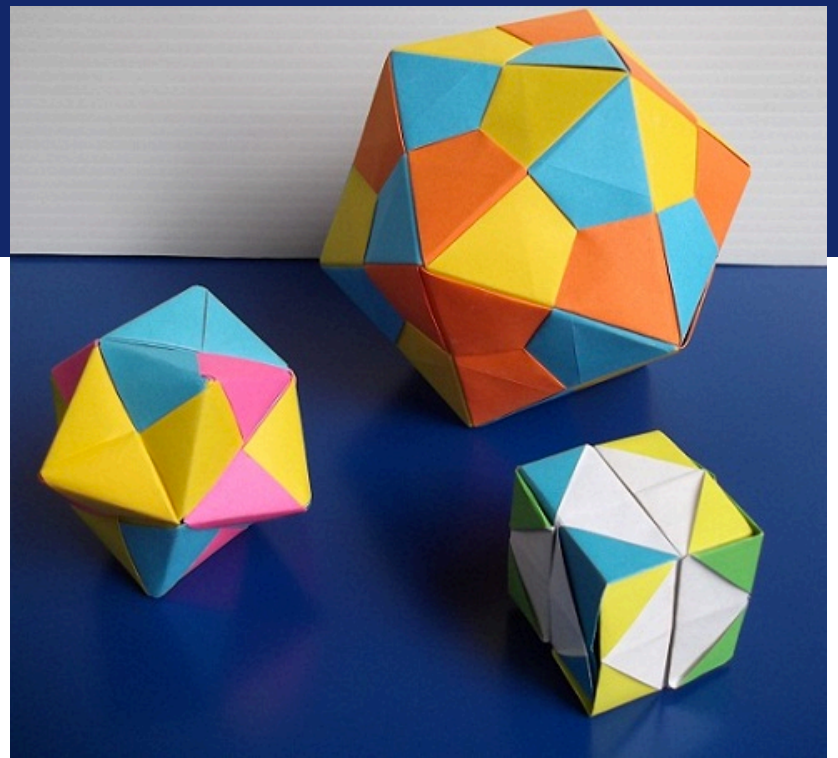


Property-Based Declarative Containers in C++



Colorado++
https://coloradoplusplus.info/

charley bay
charleyb123 at gmail dot com

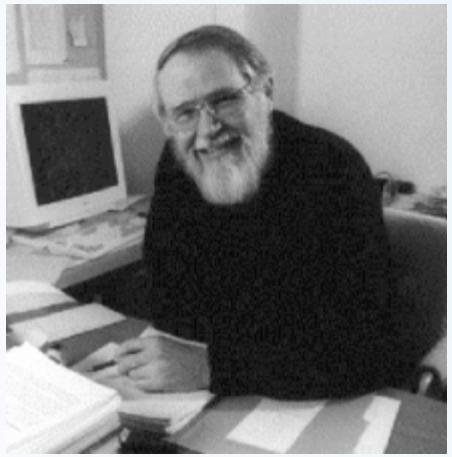
Today's Agenda

1. **Purpose** For Containers
   - **Functional** and **Non-Functional** Behavior
2. **Interacting** With Containers
   - *External:* **Interface** *(mutating and non-mutating)*
   - *Internal:* **Invariant Management**
   - *Surprising* and **unexpected container semantics**
3. Container **Interface Mechanisms**
   - **Imperative** *(iteration)* vs. **Declarative** *(properties)*
4. **Defining** a Container **Interface**
   - Separating "**interface**" from "**implementation**"
   - **Open-Sets** and **Closed-Sets**
   - **Slices**, **Sub-setting**, and **Data Reduction**
5. **Using** Containers In **Algorithms**
   - **Imperative**: Iterators and iterator invalidation
   - **Declarative**: Property extraction, type coercion, type transforms
6. **Hybrid** Containers: Both **Imperative** *And* **Declarative**
7. **Conclusions**

Controlling complexity is the essence of computer programming.

-- Brian Kernighan

Software Tools (1967), p.319 (with P.J. Plauger)



Container Purpose

What, and Why?

What Is A Container?

- What is a **container**?

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects
- What is **implied**?

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects
- What is **implied**?
  - Container **controls allocation and deallocation**
  - Container **enables *some form* of `{insert|access|erase}`**

Storage Management

Membership Lifecycle

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects
- What is **implied**?
  - Container **controls allocation and deallocation**
  - Container **enables *some form* of `{insert|access|erase}`**
- When we think about containers, we **consider**:
  1. How objects are **added and removed**
  2. How objects are **accessed**
  3. How objects are **managed** (***technical constraints*** *related to the implementation)*

Storage Management

Membership Lifecycle

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects
- What is **implied**?
  - Container **controls allocation and deallocation**
  - Container **enables *some form* of `{insert|access|erase}`**

Storage Management

Membership Lifecycle

- When we think about containers, we **consider**:
  1. How objects are **added and removed**
  2. How objects are **accessed**
  3. How objects are **managed** (***technical constraints*** *related to the implementation)*

Functional Behavior

What Is A Container?

C++ Standard *(N4800, 2019-01-21):* §21.2.1.1 *"Containers are objects that store other objects"*

- What is a **container**?
  - An **object that contains** other objects
- What is **implied**?
  - Container **controls allocation and deallocation** — *Storage Management*
  - Container **enables *some form* of `{insert|access|erase}`** — *Membership Lifecycle*
- When we think about containers, we **consider**:
  1. How objects are **added and removed**
  2. How objects are **accessed**

  Functional Behavior
  3. How objects are **managed** (***technical constraints*** *related to the implementation)*

  Non-Functional Behavior

Technical Constraints

Constraint *(def):*
An identified limitation

1. **Requirement**: A **logical** constraint
   *(allows us to **reason** about our design)*

*This is an **intentional** choice!*

2. **Technical Requirement**: A **physical** *(real)* constraint
   *(the **real-world implication** from an implementation choice)*

*This is the **reality** we must live with!*

- **Common technical constraints** within C++ containers:
  - **Access patterns** *(sequential, random, forward, etc.)*
  - **Algorithm Complexity** *(e.g., "Big-O")*
  - **Invariants** within container, examples:
    - Contiguous vs. Segmented Memory
    - Uniqueness by Value, or by Address
    - Edge-cases and "Rebuild" Events *(e.g., hash-collisions, `realloc` events)*

A Container Is A Collection Of Values

- *RECALL:* C++ is a **Value-Semantic Language**

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Foreshadowing: This will be important later

*An **Address** is also a **data object value**!*

A Container Is A Collection Of Values

- *RECALL:* C++ is a **Value-Semantic Language**

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Foreshadowing: This will be important later

*An **Address** is also a **data object value**!*

All `std::` containers perform **value semantics**

- A container-of-objects **holds the value** of those objects
- A container-of-addresses **holds the value** of those addresses
  (and does not exercise ownership of the objects at those addresses)

Container Purpose

- ***Why*** use containers?

1. As a **Design Tool**
(logical)
The collection of items have **meaning greater** than that of the individual items

Tool for scaling!

2. As **Convenience**
(physical)
Claim system resources composing **correct constructions** using **less code** than the alternative

Tool for pragmatic organization!

Container Purpose

- ***Why*** use containers?

1. As a **Design Tool**
(logical)
The collection of items have **meaning greater** than that of the individual items

Tool for scaling!

2. As **Convenience**
(physical)
Claim system resources composing **correct constructions** using **less code** than the alternative

Tool for pragmatic organization!

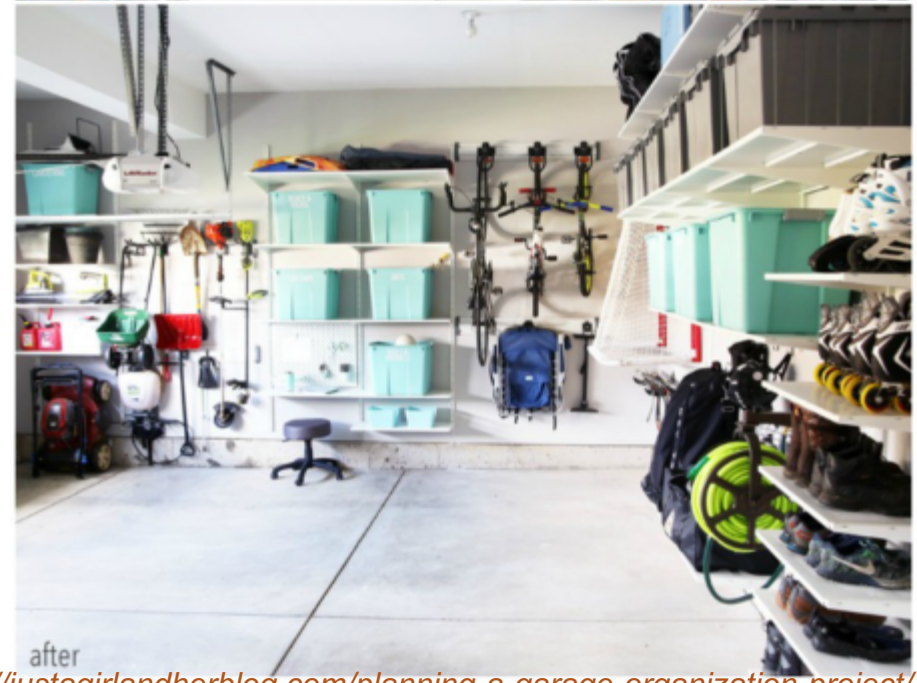
You Need Both!

1. **Design** allows our systems to **scale** based on logical *(domain)* constructs
2. **Convenience** allows us to leverage Computer Science artifacts to rely upon well-known **invariant tradeoffs**

You Need Both!

Without Containers: *"Overwhelming"*

- As a **Design Tool**:
  1. What *should* and *should not* be stored?
  2. Classification: *Group Logically*
  3. Make a *plan* *(e.g., "zones" and "priority")*
- As **Convenience**:
  1. What `std::` containers would *work best* *(for what, and where)*?
  2. *Implement*
  3. *Repair* decisions *(when you later discover you were wrong, again)*



https://justagirlandherblog.com/planning-a-garage-organization-project/

Container As A *Design Tool*

```
struct HistogramData
{
    std::vector<int> bins_;
};
```

1 As A **Design Tool** *(logical)* The collection of items have **meaning greater** than that of the individual items

Container As A *Design Tool*

As A **Design Tool** *(logical)*
1 The collection of items have **meaning greater** than that of the individual items

```
struct HistogramData
{
    std::vector<int> bins_;
};
```

- **Enforcement** of domain-specific invariants *(examples):*
  - Each “bin” may have **domain-specific value-restrictions** and “trap” values
  - Each “bin” has an **ordered position** within the collection that implies its “meaning” *(based on position)*
  - **Enforcement of rules** governing “bin” {`insert`} and {`erase`}

Example: Reserve value for “scale-exceeded”

Example: Ordered bin distribution is implicitly “linear” or “logarithmic”

Example: Might only support “powers-of-two” for number-of-bins in container

Container As A *Design Tool*

As A **Design Tool** *(logical)*
1 The collection of items have **meaning greater** than that of the individual items

```
struct HistogramData
{
    std::vector<int> bins_;
};
```

- **Enforcement** of domain-specific invariants *(examples):*
  - Each “bin” may have **domain-specific value-restrictions** and “trap” values
  - Each “bin” has an **ordered position** within the collection that implies its “meaning” *(based on position)*
  - **Enforcement of rules** governing “bin” {`insert`} and {`erase`}

Example: Reserve value for “scale-exceeded”

Example: Ordered bin distribution is implicitly “linear” or “logarithmic”

Example: Might only support “powers-of-two” for number-of-bins in container

Example: Functions accept only “bin-counts”, and not “integers”

- Enables **aggregation** of domain state-and-logic *(examples):*
  - Domain-specific functions operate only on “bins” to **leverage the C++ type system**
  - Domain-specific operations on “bins” can be **defined within** domain-specific **container**

Container As A *Design Tool*

```
struct HistogramData
{
    std::vector<int> bins_;
};
```

- **Enforcement** of domain-specific invariants *(examples):*
  - Each “bin” may have **domain-specific value-restrictions** and “trap” values
  - Each “bin” has an **ordered position** within the collection that implies its “meaning” *(based on position)*
  - **Enforcement of rules** governing “bin” {`insert`} and {`erase`}

Example: Reserve value for “scale-exceeded”

Example: Ordered bin distribution is implicitly “linear” or “logarithmic”

Example: Might only support “powers-of-two” for number-of-bins in container

1 As A **Design Tool** *(logical)*
The collection of items have **meaning greater** than that of the individual items

Example: Functions accept only “bin-counts”, and not “integers”

- Enables **aggregation** of domain state-and-logic *(examples):*
  - Domain-specific functions operate only on “bins” to **leverage the C++ type system**
  - Domain-specific operations on “bins” can be **defined within** domain-specific **container**

A Domain-Specific Container is a **logical aggregation**

- Is **state-with-rules** implied by the domain
- Is **abstraction** suitable for ***reasoning*** within the domain

Container As A *Convenience*

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

2

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

Container As A *Convenience*

2

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

Local abstraction documenting "key"

Artifact-of-behavior: Container enforces uniqueness-by-key

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

Container As A *Convenience*

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

2

Local abstraction documenting "key"

Artifact-of-behavior: Container enforces uniqueness-by-key

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

```
struct MyConnection { /*...*/ };
struct MyConnections {
  static constexpr const uint16_t MAX_NUM_CONNECTIONS_ = 1024;
  std::array<MyConnection, MAX_NUM_CONNECTIONS_> my_connections_;
};
```

Container As A *Convenience*

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

2

Local abstraction documenting "key"

Artifact-of-behavior: Container enforces uniqueness-by-key

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

Local establishment of size requirements (artifact)

Claim contiguous memory supporting fast-access

```
struct MyConnection { /*...*/ };
struct MyConnections {
  static constexpr const uint16_t MAX_NUM_CONNECTIONS_ = 1024;
  std::array<MyConnection, MAX_NUM_CONNECTIONS_> my_connections_;
};
```

Container As A *Convenience*

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

2

Local abstraction documenting "key"

Artifact-of-behavior: Container enforces uniqueness-by-key

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

Local establishment of size requirements (artifact)

Claim contiguous memory supporting fast-access

```
struct MyConnection { /*...*/ };
struct MyConnections {
  static constexpr const uint16_t MAX_NUM_CONNECTIONS_ = 1024;
  std::array<MyConnection, MAX_NUM_CONNECTIONS_> my_connections_;
};
```

1. Code is **reused**
2. **Invariants** are **enforced**
3. Abstraction *(behavior)* is **well-understood**

Container As A *Convenience*

As **Pragmatic Organization**
(physical)
Claim system resources
composing **correct constructions**
using **less code**
than the alternative

2

Local abstraction documenting "key"

Artifact-of-behavior: Container enforces uniqueness-by-key

```
struct MyColorValue { /*...*/ };
struct MyColors {
  using MyColorName = std::string;
  // unique-by-name
  std::unordered_map<MyColorName, MyColorValue> my_colors_;
};
```

Local establishment of size requirements (artifact)

Claim contiguous memory supporting fast-access

```
struct MyConnection { /*...*/ };
struct MyConnections {
  static constexpr const uint16_t MAX_NUM_CONNECTIONS_ = 1024;
  std::array<MyConnection, MAX_NUM_CONNECTIONS_> my_connections_;
};
```

1. Code is **reused**
2. **Invariants** are **enforced**
3. Abstraction *(behavior)* is **well-understood**

A Domain-Specific Container
is a **physical aggregation**

- **Claims** a collection of **system resources** with simple and elegant patterns
- Leverages **re-use of invariants** for well-known constructs *(enabling reasoning of behavior)*

Container Behavior

Behavior: Functional and Non-Functional

Functional vs. Non-Functional

- **Functional Behavior** is that which serves the container's **purpose**
  - **Is a "specification"** between inputs-and-outputs
  - **Is "why"** you employed that container

Trivia: IEEE-Std 830-1993 lists 13 non-functional software requirements:

1. Performance
2. Interface
3. Operational
4. Resource
5. Verification
6. Acceptance
7. Documentation
8. Security
9. Portability
10. Quality
11. Reliability
12. Maintainability
13. Safety

Other considerations might be:

14. Accessibility
15. Capacity *(current and forecast)*
16. Compliance
17. Disaster Recovery
18. Efficiency
19. Effectiveness
20. Extensibility
21. Fault Tolerance
22. Interoperability
23. Privacy
24. Resilience
25. Response Time
26. Robustness
27. Scalability
28. Stability
29. Supportability
30. Testability

…and more at:
https://en.wikipedia.org/wiki/Non-functional_requirement

Functional vs. Non-Functional

- **Functional Behavior** is that which serves the container's **purpose**
  - **Is a "specification"** between inputs-and-outputs
  - **Is "why"** you employed that container
- **Non-Functional Behavior** is *criteria* to judge the operational result *(these are):*
  - "**quality attributes**", "**constraints**", "**technical requirements**"

Trivia: IEEE-Std 830-1993 lists 13 non-functional software requirements:

1. Performance
2. Interface
3. Operational
4. Resource
5. Verification
6. Acceptance
7. Documentation
8. Security
9. Portability
10. Quality
11. Reliability
12. Maintainability
13. Safety

Other considerations might be:

14. Accessibility
15. Capacity *(current and forecast)*
16. Compliance
17. Disaster Recovery
18. Efficiency
19. Effectiveness
20. Extensibility
21. Fault Tolerance
22. Interoperability
23. Privacy
24. Resilience
25. Response Time
26. Robustness
27. Scalability
28. Stability
29. Supportability
30. Testability

…and more at:
https://en.wikipedia.org/wiki/Non-functional_requirement

Functional vs. Non-Functional

- **Functional Behavior** is that which serves the container's **purpose**
  - **Is a "specification"** between inputs-and-outputs
  - **Is "why"** you employed that container
- **Non-Functional Behavior** is *criteria* to judge the operational result *(these are):*
  - "**quality attributes**", "**constraints**", "**technical requirements**"

Functional: **Necessary** Behavior ← ***What*** *it should do*

Non-Functional: **Criteria to judge behavior** in particular conditions *(is **not** specific behavior)* ← ***Constraints*** *on* ***How*** *it should do*

Quality Of Implementation *("QoI")* discussions relate to Non-Functional Attributes

Trivia: IEEE-Std 830-1993 lists 13 non-functional software requirements:

1. Performance
2. Interface
3. Operational
4. Resource
5. Verification
6. Acceptance
7. Documentation
8. Security
9. Portability
10. Quality
11. Reliability
12. Maintainability
13. Safety

Other considerations might be:

14. Accessibility
15. Capacity *(current and forecast)*
16. Compliance
17. Disaster Recovery
18. Efficiency
19. Effectiveness
20. Extensibility
21. Fault Tolerance
22. Interoperability
23. Privacy
24. Resilience
25. Response Time
26. Robustness
27. Scalability
28. Stability
29. Supportability
30. Testability

…and more at:
https://en.wikipedia.org/wiki/Non-functional_requirement

Functional vs. Non-Functional

- **Functional Behavior** is that which serves the container's **purpose**
  - **Is a "specification"** between inputs-and-outputs
  - **Is "why"** you employed that container
- **Non-Functional Behavior** is *criteria* to judge the operational result *(these are):*
  - "**quality attributes**", "**constraints**", "**technical requirements**"

Functional: **Necessary** Behavior ← ***What*** *it should do*

Non-Functional: **Criteria to judge behavior** in particular conditions *(is **not** specific behavior)* ← ***Constraints*** *on* ***How*** *it should do*

Quality Of Implementation *("QoI")* discussions relate to Non-Functional Attributes

Trivia: IEEE-Std 830-1993 lists 13 non-functional software requirements:

1. Performance
2. Interface
3. Operational
4. Resource
5. Verification
6. Acceptance
7. Documentation
8. Security
9. Portability
10. Quality
11. Reliability
12. Maintainability
13. Safety

Other considerations might be:

14. Accessibility
15. Capacity *(current and forecast)*
16. Compliance
17. Disaster Recovery
18. Efficiency
19. Effectiveness
20. Extensibility
21. Fault Tolerance
22. Interoperability
23. Privacy
24. Resilience
25. Response Time
26. Robustness
27. Scalability
28. Stability
29. Supportability
30. Testability

…and more at: https://en.wikipedia.org/wiki/Non-functional_requirement

Distinguishing between Functional and Non-Functional **can be a "Gray Area"**, may depend upon:

- Level of **detail required** *(what you **really** care about)*
- The **degree of trust** assumed *(requirements are **constraints**)*

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

What *it should do (example: Device only supports contiguous memory copy)*

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* {`insert`} may require `realloc`

What *it should do (example: Device only supports contiguous memory copy)*

Constraints *on* ***How*** *it should do*

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* `{insert}` may require `realloc`

Functional: Require **content-based lookup** *(e.g., "index-by-hash")*

***What** it should do (example: Device only supports contiguous memory copy)*

***Constraints** on **How** it should do*

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* `{insert}` may require `realloc`

Functional: Require **content-based lookup** *(e.g., "index-by-hash")*

Non-Functional: *(technical constraint)* different keys may result in **hash-collisions**

What *it should do (example: Device only supports contiguous memory copy)*

Constraints *on* ***How*** *it should do*

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* `{insert}` may require `realloc`

Functional: Require **content-based lookup** *(e.g., "index-by-hash")*

Non-Functional: *(technical constraint)* different keys may result in **hash-collisions**

Functional: Require **data layout** that a given algorithm can access **efficiently**

What *it should do (example: Device only supports contiguous memory copy)*

Constraints *on* ***How*** *it should do*

Functional Behavior

- Functional and Non-Functional Examples:

__What__ it should do (example: Device only supports contiguous memory copy)

__Constraints__ on __How__ it should do

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* `{insert}` may require `realloc`

Functional: Require **content-based lookup** *(e.g., "index-by-hash")*

Non-Functional: *(technical constraint)* different keys may result in **hash-collisions**

Functional: Require **data layout** that a given algorithm can access **efficiently**

Non-Functional: *(technical constraint)* A different access pattern from a different algorithm will be **inefficient**

Functional Behavior

- Functional and Non-Functional Examples:

Functional: Require **dynamically-sized contiguous** memory

Non-Functional: *(technical constraint)* `{insert}` may require `realloc`

Functional: Require **content-based lookup** *(e.g., "index-by-hash")*

Non-Functional: *(technical constraint)* different keys may result in **hash-collisions**

Functional: Require **data layout** that a given algorithm can access **efficiently**

Non-Functional: *(technical constraint)* A different access pattern from a different algorithm will be **inefficient**

- **Summary**: Functional, and Non-Functional Behavior

We select a Computer Science artifact based on functional requirements

We live with the implications *(technical constraints)* implied by that artifact

What *it should do (example: Device only supports contiguous memory copy)*

Constraints *on* ***How*** *it should do*

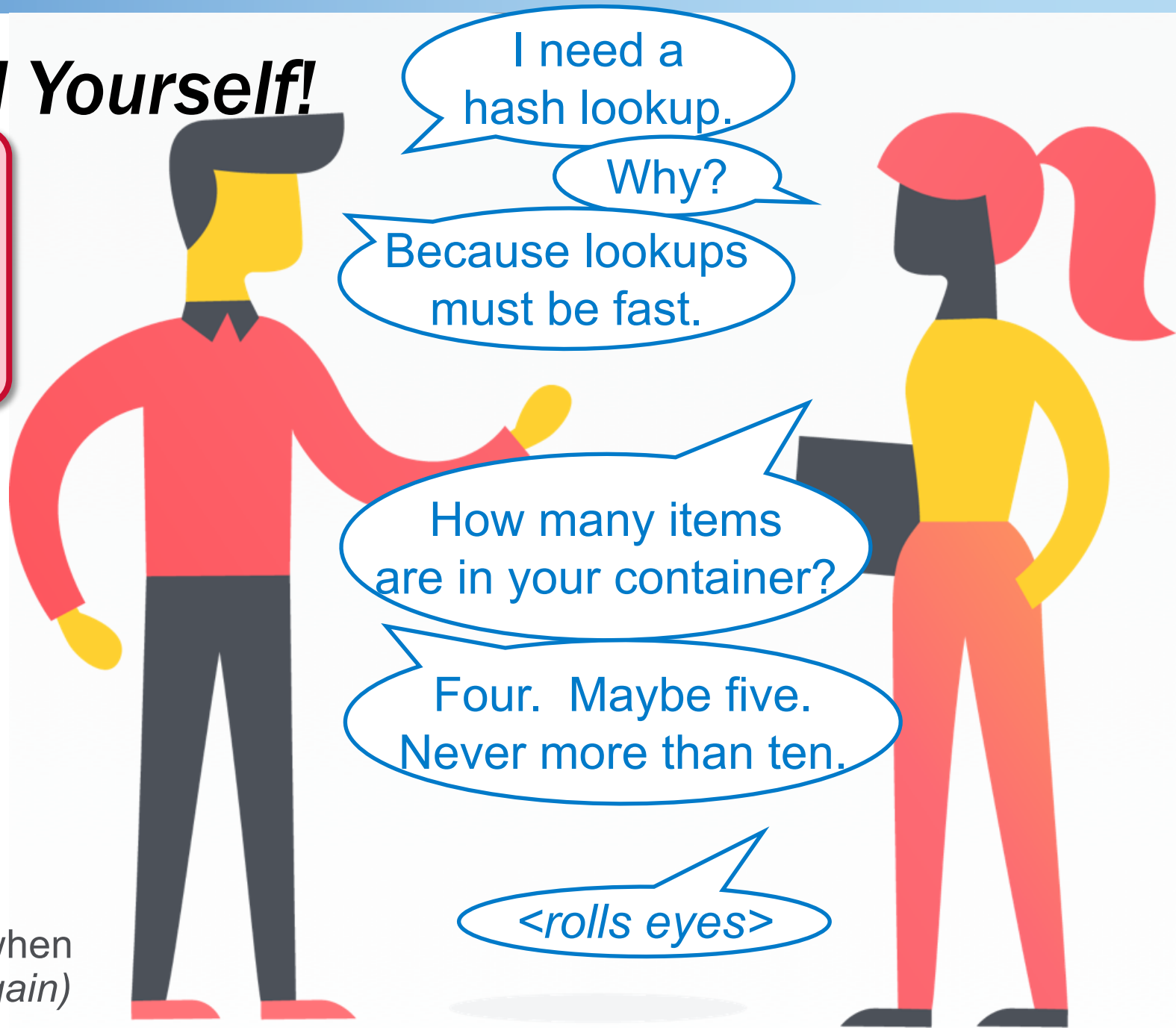
Design *is "balancing both"*

Functional? *Fool Yourself!*

Watch Out:
We can "fool ourselves" to rationalize *(necessary)* "functional" behavior.

- ***Consider:***
  - Access patterns
  - Scale
  - Edge-cases
- ***Verify:***
  - Through Measured Behavior at different system loads
- ***Change:***
  - Select different implementation when you discover you were wrong *(again)*

Would-be-nice *if this was* ***low-cost****!*



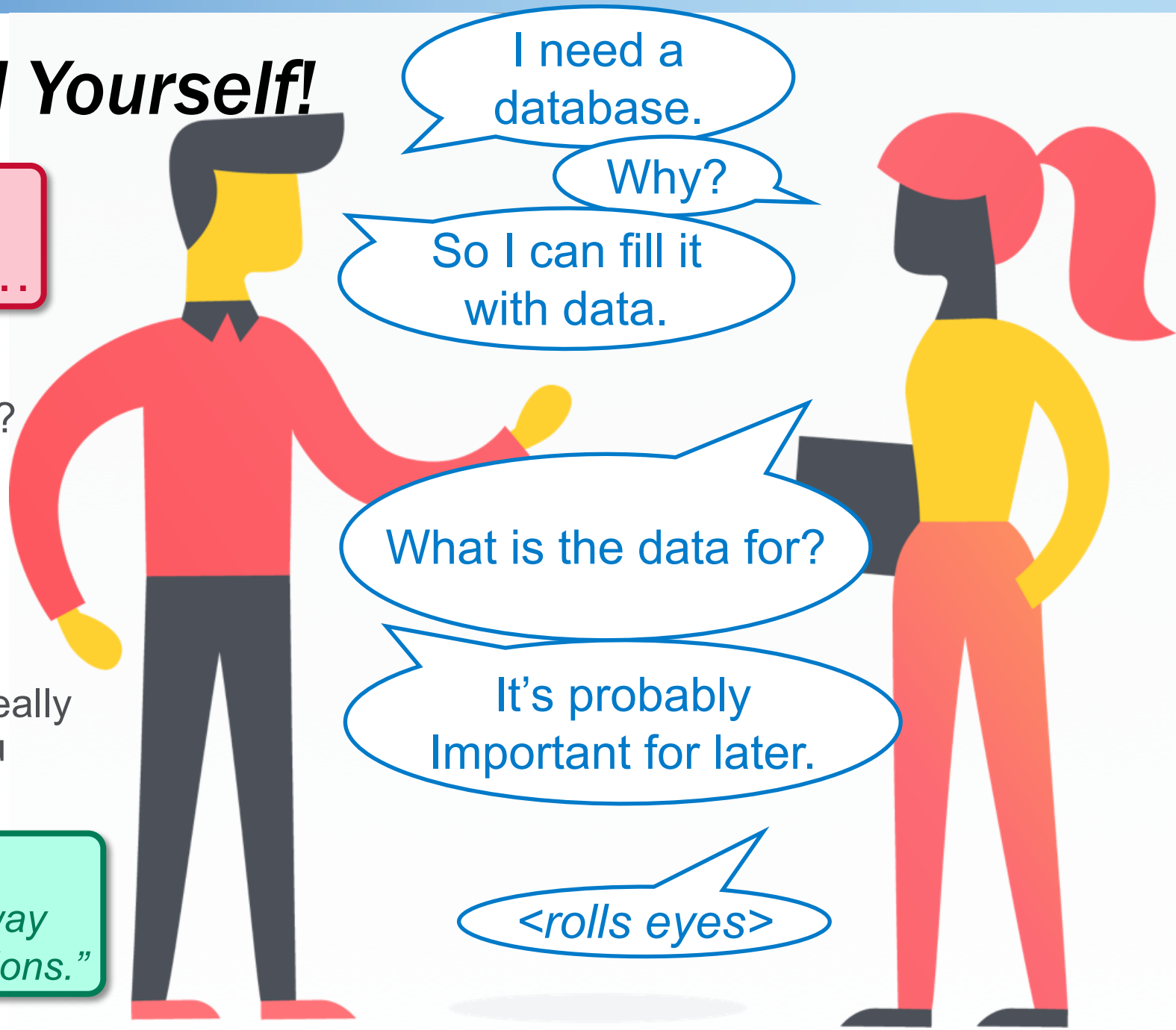
Functional? *Fool Yourself!*

1990's:
Overheard in the hallway…

- ***Consider:***
  - **What** do you ***really*** care about?
- ***Verify:***
  - Were you ***correct***?
- ***Change:***
  - Prioritize what turned out to be really important when you discover you were wrong *(again)*

Would-be-nice
if this was ***low-cost****!*

Two Decades Later:
"I need to access my data in a way that lets me answer specific questions."



You Were Probably Wrong

Your "guess" between "Functional" and "Non-Functional" **is probably wrong**

- *Given:* The "`size`" of a container depends on the objects it contains

- **Which** of the following **is *actually* important** for your use-case?
  - `size`, as *"item-count"*: The number of elements contained
  - `size`, as *"memory requirements":*
    - The memory allocated *(i.e., "reserved")*
    - The memory in-use to hold elements
  - `size`, as *"unique-item-count":* Container may hold object duplicates

***WARNING: Functional Requirement May Change**, example:*

1. You "guessed" you needed ***O(1)*** for `insert`
2. In reality, ***O(n)*** was fine for `insert`
3. You discover you really do need ***O(1)*** for element `read`

Responding To System Changes

- Computer Science suggests approaches to enable later changes when you discover your “guess” was wrong *(again)*

Hint: **Computer Science** suggests separating:

- “**interface**” *(what you use)*

…from

- “**implementation**” *(how that is provided)*

Interacting With Containers

Interfaces, Invariants, and Surprises

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

2. *Internal:* **Invariant Management**
   - Responding to **external** demands *(interface support)*
   - Responding to **internal** demands *(invariant maintenance)*

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

Smells a lot like ***Functional Behavior***

2. *Internal:* **Invariant Management**
   - Responding to **external** demands *(interface support)*
   - Responding to **internal** demands *(invariant maintenance)*

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

Smells a lot like ***Functional Behavior***

2. *Internal:* **Invariant Management**
   - Responding to **external** demands *(interface support)*
   - Responding to **internal** demands *(invariant maintenance)*

Smells a lot like ***Non-Functional Behavior***

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

Smells a lot like ***Functional Behavior***

2. *Internal:* **Invariant Management**
   - Responding to **external** demands *(interface support)*
   - Responding to **internal** demands *(invariant maintenance)*

Smells a lot like ***Non-Functional Behavior***

3. **Surprises**
   - **Edge Cases**
   - **Unexpected semantics** and behavior

Interacting With Containers

- **Container interaction** is comprised of *(in descending priority):*

1. *External:* **Interfaces**
   - **Non-Mutating** *(`const`)*
   - **Mutating** *(non-`const`)*

Smells a lot like ***Functional Behavior***

2. *Internal:* **Invariant Management**
   - Responding to **external** demands *(interface support)*
   - Responding to **internal** demands *(invariant maintenance)*

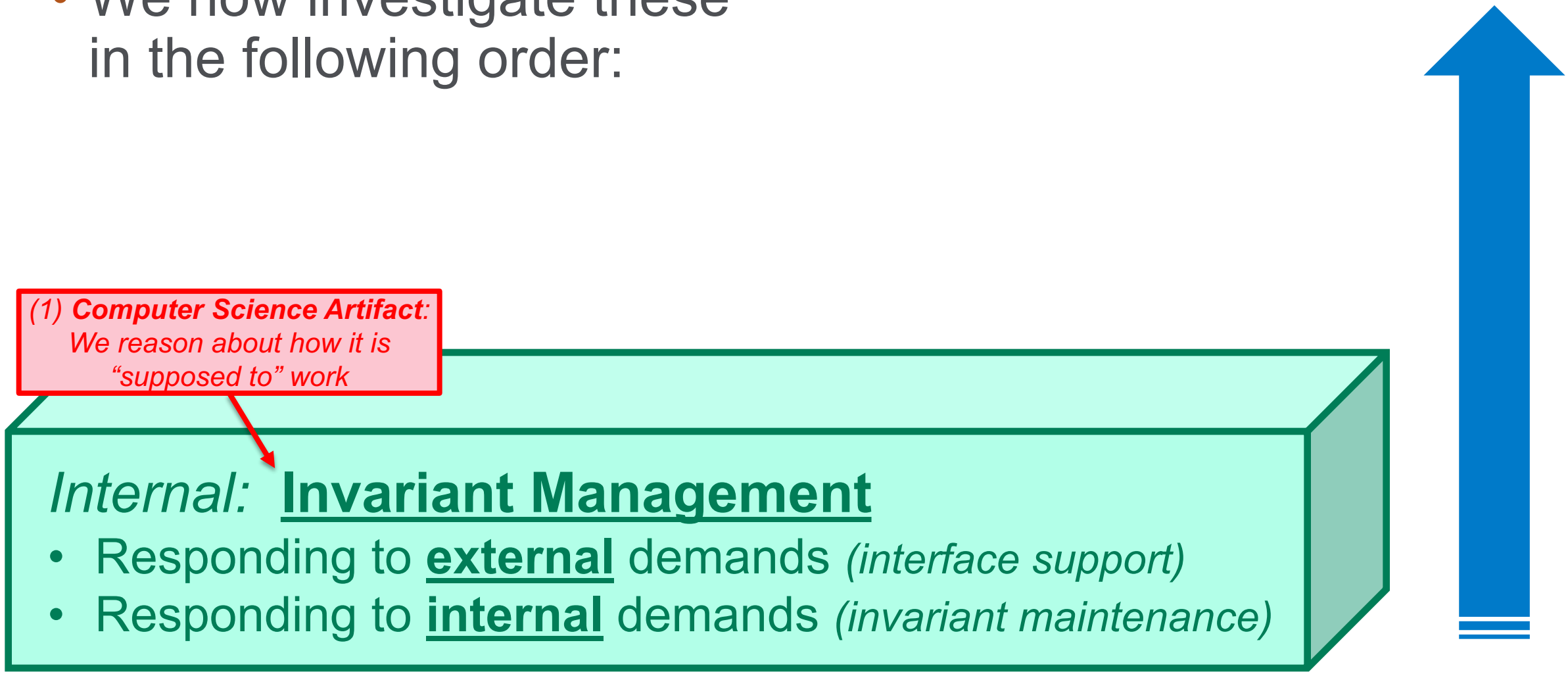
Smells a lot like ***Non-Functional Behavior***

3. **Surprises**
   - **Edge Cases**
   - **Unexpected semantics** and behavior

Smells a lot like ***Impedence Mismatch*** *between real-world use and Functional or Non-Functional Behavior*

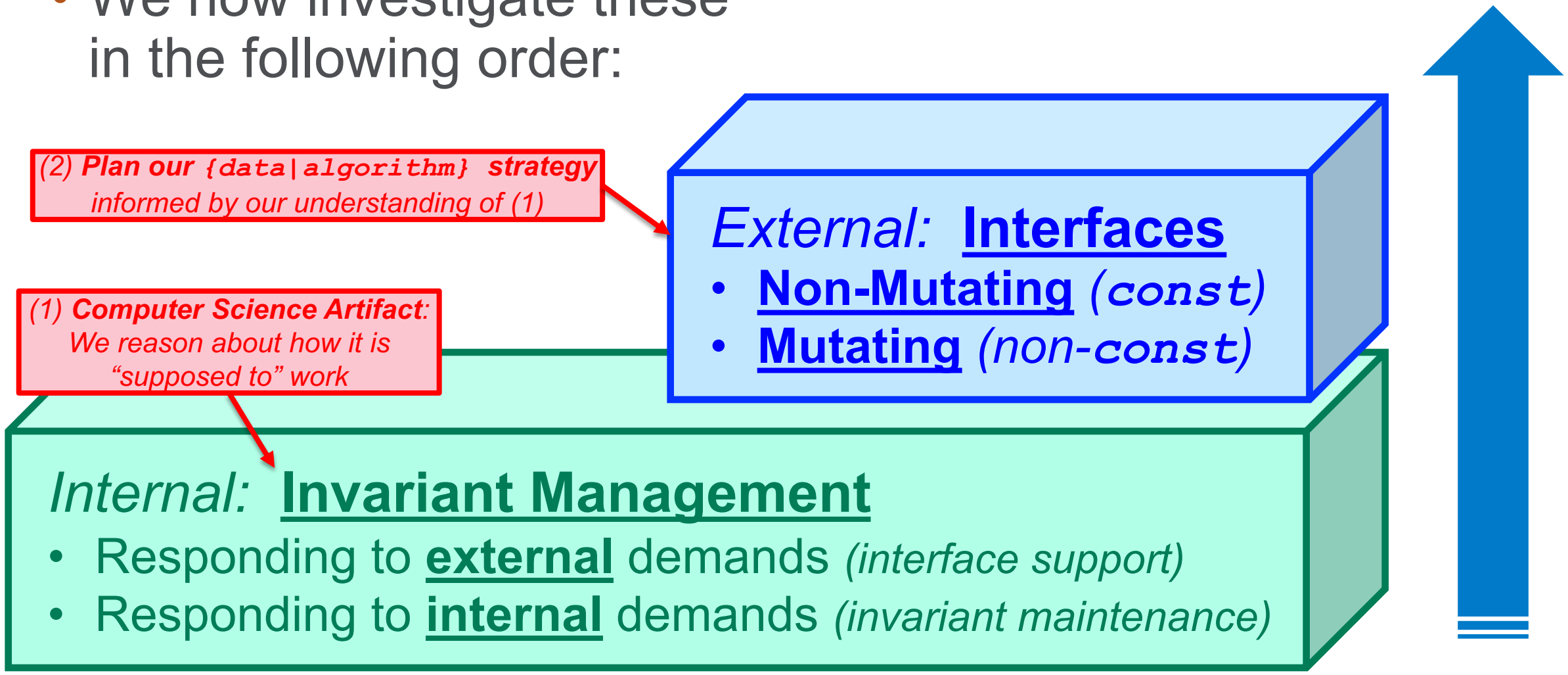
Building Our Understanding

- We now investigate these in the following order:



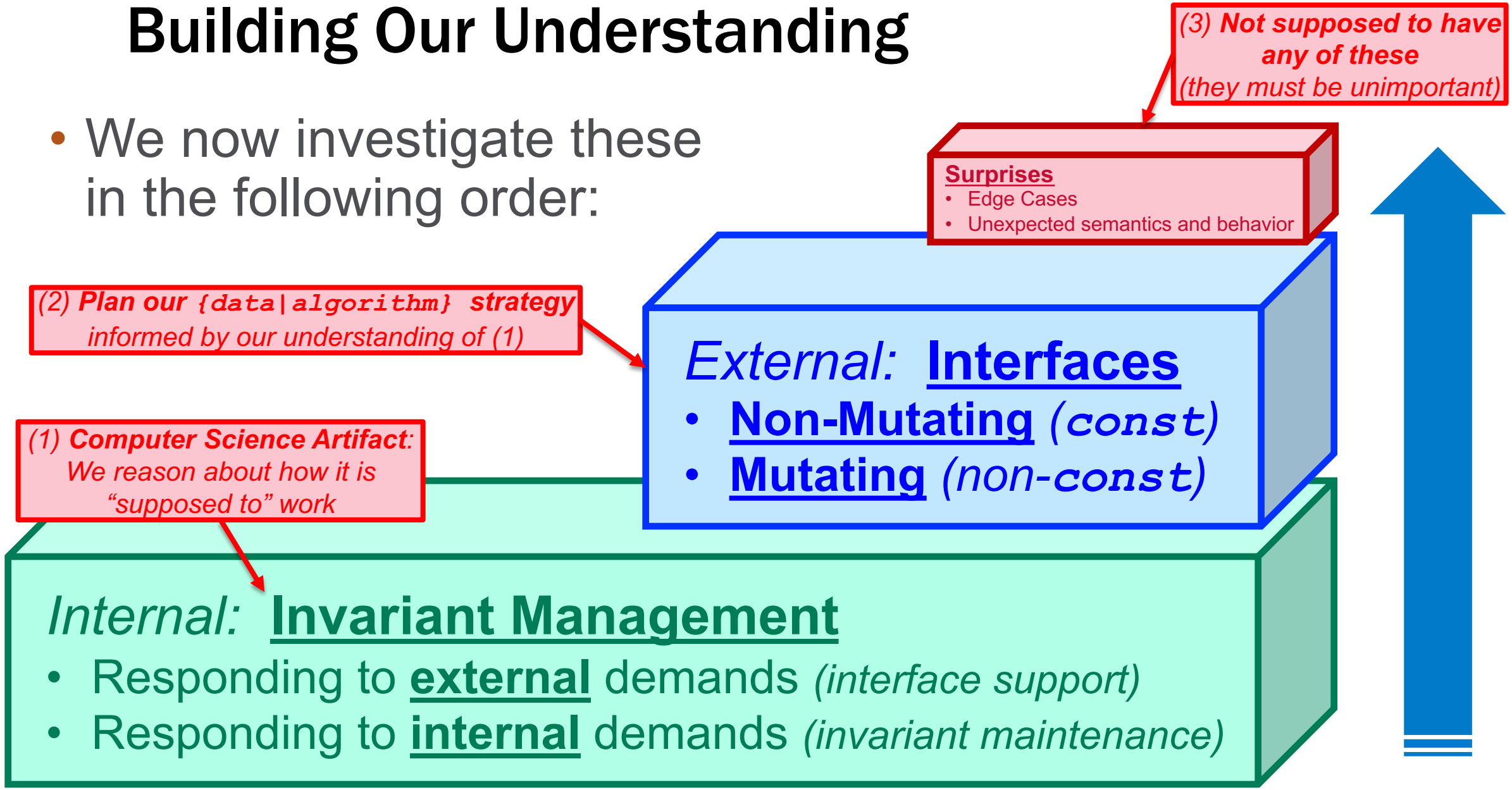
Building Our Understanding

- We now investigate these in the following order:



Building Our Understanding

- We now investigate these in the following order:



C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

Associative Containers

Sorted *(and ordered!)* data structures with **fast search**: O(log n)

```
std::set       //            keys, sorted by keys, keys are unique
std::multiset  //            keys, sorted by keys
std::map       // key-value pairs, sorted by keys, keys are unique
std::multimap  // key-value pairs, sorted by keys
```

C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

Associative Containers

Sorted *(and ordered!)* data structures with **fast search**: O(log n)

```
std::set       //                keys, sorted by keys, keys are unique
std::multiset  //                keys, sorted by keys
std::map       // key-value pairs, sorted by keys, keys are unique
std::multimap  // key-value pairs, sorted by keys
```

"multi" – ***multiple entries for the same key*** *are permitted*

C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

Associative Containers

Sorted *(and ordered!)* data structures with **fast search**: O(log n)

```
std::set       //                 keys, sorted by keys, keys are unique
std::multiset  //                 keys, sorted by keys
std::map       // key-value pairs, sorted by keys, keys are unique
std::multimap  // key-value pairs, sorted by keys
```

*"multi" – **multiple entries for the same key** are permitted*

Unordered Associative Containers

Hashed data structures with **fast search**: O(1) *amortized*, O(n) *worst-case*

```
std::unordered_set       //                 keys, hashed by keys, keys are unique
std::unordered_multiset  //                 keys, hashed by keys
std::unordered_map       // key-value pairs, hashed by keys, keys are unique
std::unordered_multimap  // key-value pairs, hashed by keys
```

C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

Container Adapters

Interface over **Sequential** containers

```
std::stack          // LIFO
std::queue          // FIFO
std::priority_queue // O(1)access, O(n)insert
```

Associative Containers

Sorted *(and ordered!)* data structures with **fast search**: O(log n)

```
std::set      //                keys, sorted by keys, keys are unique
std::multiset //                keys, sorted by keys
std::map      // key-value pairs, sorted by keys, keys are unique
std::multimap // key-value pairs, sorted by keys
```

"multi" – ***multiple entries for the same key*** *are permitted*

Unordered Associative Containers

Hashed data structures with **fast search**: O(1) *amortized*, O(n) *worst-case*

```
std::unordered_set      //                keys, hashed by keys, keys are unique
std::unordered_multiset //                keys, hashed by keys
std::unordered_map      // key-value pairs, hashed by keys, keys are unique
std::unordered_multimap // key-value pairs, hashed by keys
```

C++ Standard Containers

`std::` containers are **classified by concept**

Sequence Containers

Sequential *(ordered)* access to elements

```
std::array          //  static contiguous array
std::vector         // dynamic contiguous array
std::deque          // double-ended queue
std::forward_list   // singly-linked list
std::list           // doubly-linked list
```

Container Adapters

Interface over **Sequential** containers

```
std::stack          // LIFO
std::queue          // FIFO
std::priority_queue // O(1)access, O(n)insert
```

Associative Containers

Sorted *(and ordered!)* data structures with **fast search**: O(log n)

```
std::set      //                keys, sorted by keys, keys are unique
std::multiset //                keys, sorted by keys
std::map      // key-value pairs, sorted by keys, keys are unique
std::multimap // key-value pairs, sorted by keys
```

Views

View over **Contiguous** objects

```
std::span // View over contiguous
          // object sequence
```

"multi" – ***multiple entries for the same key*** *are permitted*

Unordered Associative Containers

Hashed data structures with **fast search**: O(1) *amortized*, O(n) *worst-case*

```
std::unordered_set      //                keys, hashed by keys, keys are unique
std::unordered_multiset //                keys, hashed by keys
std::unordered_map      // key-value pairs, hashed by keys, keys are unique
std::unordered_multimap // key-value pairs, hashed by keys
```

Trivia!

`std::string` meets the requirements of a **Sequence** Container

Common Container Interfaces

- A `std::` container has:
  - `empty()`
    - …Returns `true` if **no elements** in container
  - `size()` *(except `std::forward_list`)*
    - …**Number of elements** in container *(example):* `a.size()`
  - `begin()` , `end()` , *…(and others)*
    - …**Iterators** *(**positional handles** to member-items, **like a "cursor"**)*
  - `operator==()` , `operator!=()`
    - …**Equivalence** *(example):* `a==b`
  - `swap()`
    - …**Swap** *(example):* `a.swap(b)`

This is **not a big list**.
These are primitive abstractions.

- Also:
  - **Some** have {`emplace_front`|`push_front`}
  - **Sequential** containers have:
    - {`push_back`|`emplace_back`}
    - {`insert`} and {`emplace`} at iterator

`std::` Container Strategies And Tradeoffs

- Studying the **C++ Standard Libraries** is a great way to learn Computer Science **Data Structures** and **Algorithms**

The `std::`

`{containers|algorithms|concepts}`

are **Computer Science Artifacts**:

- **Generically Applicable** *(by definition)*
- Leverage **Well-Known Strategies**
- Tradeoffs are **Well-Considered**
- **Guaranteed Behavior** is Specified
- Are **Highly Composable**

1. ***Ready-To-Use*** *Directly*
2. ***Suitable*** *for Many Domains (probably yours also)*

Attributes For `std::` Container Types

- Container types are **classified by *concept***
  - **Node-based** containers *(e.g., `std::list`)*
  - **Contiguous** containers *(e.g., `std::vector`)*
  - Both **node and contiguous** *(e.g., `std::deque`)*
  - **Heap or Stack** containers *(e.g., `std::vector` and `std::array` are both contiguous)*

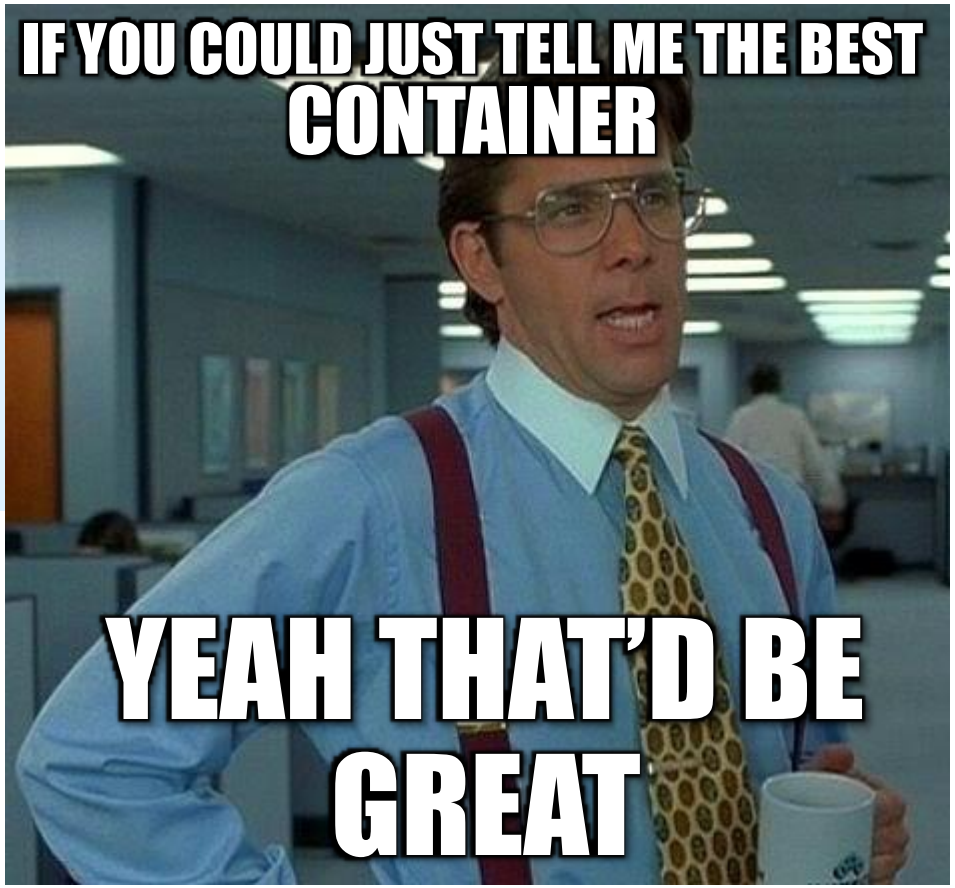
- A `std::` container:
  - **Has** a set of **typedefs** *(`value_type`, `iterator`, `const_iterator`, etc.)*
  - **May have** an **allocator** *(except `std::array`)*
    - **Allocator Job**: Construct nodes containing aligned buffers, and then call `allocator_traits<allocator_type>::rebind_traits<U>::construct` to place element into the buffer
  - Can be **constructed from** any of:
    - Nothing *(except `std::array`)*
    - Two input or forward iterators *(to a compatible value type)*
    - An initializer list

The “Best” Container

- **No “Best”** Container exists

```
std::array          // fast access but fixed number of elements
std::vector         // fast access but inefficient {insert|erase}
std::deque          // efficient {insert|erase} at {front|back}
std::forward_list   // efficient {insert|erase} in middle
std::list           // efficient {insert|erase} in middle
```

A given domain-specific type likely requires **different container tradeoffs** when used **in different algorithms**



How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior

How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior
- The **interface is implied** based on **invariants** within the container
  - `{insert|access|erase}` are **defined** in terms of the **container strategy**
  - These comprise the **public interface**

How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior
- The **interface is implied** based on **invariants** within the container
  - {`insert`|`access`|`erase`} are **defined** in terms of the **container strategy**
  - These comprise the **public interface**
- The interface "**leaks" implementation details**; depending on the container strategy:
  - Some {`insert`|`access`|`erase`} options **are "encouraged"**
  - Some {`insert`|`access`|`erase`} options **are expensive**, or **not possible**
  - **Strategy-specific interface may also be provided** *(e.g.,* `bucket()`, `bucket_size()`, `bucket_count()`, `load_factor()`, `max_load_factor()`, `rehash()`, *etc.)*

How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior

- The **interface is implied** based on **invariants** within the container
  - {`insert`|`access`|`erase`} are **defined** in terms of the **container strategy**
  - These comprise the **public interface**

Functional Behavior

- The interface "**leaks" implementation details**; depending on the container strategy:
  - Some {`insert`|`access`|`erase`} options **are "encouraged"**
  - Some {`insert`|`access`|`erase`} options **are expensive**, or **not possible**
  - **Strategy-specific interface may also be provided** *(e.g.,* ***`bucket()`, `bucket_size()`, `bucket_count()`, `load_factor()`, `max_load_factor()`, `rehash()`*** *, etc.)*

How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior

- The **interface is implied** based on **invariants** within the container
  - {`insert`|`access`|`erase`} are **defined** in terms of the **container strategy**
  - These comprise the **public interface**

Functional Behavior

- The interface "**leaks" implementation details**; depending on the container strategy:
  - Some {`insert`|`access`|`erase`} options **are "encouraged"**
  - Some {`insert`|`access`|`erase`} options **are expensive**, or **not possible**
  - **Strategy-specific interface may also be provided** *(e.g.,* `bucket()`, `bucket_size()`, `bucket_count()`, `load_factor()`, `max_load_factor()`, `rehash()`, *etc.)*

Non-Functional Behavior

How Do `std::` Containers Work?

- A **strategy is defined** *(with an associated asymptotic complexity)* that allows us to reason about container behavior

- The **interface is implied** based on **invariants** within the container
  - `{insert|access|erase}` are **defined** in terms of the **container strategy**
  - These comprise the **public interface**

Functional Behavior

- The interface "**leaks" implementation details**; depending on the container strategy:
  - Some `{insert|access|erase}` options **are "encouraged"**
  - Some `{insert|access|erase}` options **are expensive**, or **not possible**
  - **Strategy-specific interface may also be provided** *(e.g., `bucket()`, `bucket_size()`, `bucket_count()`, `load_factor()`, `max_load_factor()`, `rehash()`, etc.)*

Non-Functional Behavior

Very "Leaky" Abstraction:

By definition *(and by necessity!)*, the `std::` container interface **"leaks" implementation details** through its API

`std::` Containers Are Leaky Abstractions

- A "**Universal Generic" container** that attempts to "not-leak" its details is possible. *(Other programming languages sometimes try to provide this.)*

Leaky Abstraction *(def):*

1. One that leaks details that are supposed to be abstracted away
2. The undesired exposure of implementation details

`std::` Containers Are Leaky Abstractions

- A "**Universal Generic" container** that attempts to "not-leak" its details is possible. *(Other programming languages sometimes try to provide this.)*
- However, most of the time, **we would not want it**:
  - A "universal generic" container would be **inefficient most of the time**
  - A "universal generic" container **disallows** us to **reason about the Abstract Machine** *(some programs could not be written)*
  - C++ has **a stronger Type System**:
    - We want to propagate **operations** from **domain-specific types** *(user-type operations can be injected directly into `std::` library behavior)*
    - We want **compiler-enforcement of type rules** *(a form of Program Verification)*

Leaky Abstraction *(def):*

1. One that leaks details that are supposed to be abstracted away
2. The undesired exposure of implementation details

`std::` Containers Are Leaky Abstractions

- A "**Universal Generic" container** that attempts to "not-leak" its details is possible. *(Other programming languages sometimes try to provide this.)*
- However, most of the time, **we would not want it**:
  - A "universal generic" container would be **inefficient most of the time**
  - A "universal generic" container **disallows** us to **reason about the Abstract Machine** *(some programs could not be written)*
  - C++ has **a stronger Type System**:
    - We want to propagate **operations** from **domain-specific types** *(user-type operations can be injected directly into `std::` library behavior)*
    - We want **compiler-enforcement of type rules** *(a form of Program Verification)*

Leaky Abstraction *(def):*
1. One that leaks details that are supposed to be abstracted away
2. The undesired exposure of implementation details

The **std::** containers **necessarily "leak"** implementation details:
- These are **low-level abstractions** that provide guarantees, from which we *(reason!) to* build our own data structures and algorithms
- These are **described sufficiently** *(specified!)* to enable *(efficient!)* interaction and composition with algorithms and other containers

GOOD!
…so we can be ***efficient!*** *(and* ***fast!****)*

Container Complexity

`std::` containers are **necessarily "leaky"** *because* they provide

Complexity *Guarantees*

…where that complexity **directly results** from the **container implementation**

Container Complexity

`std::` containers are **necessarily "leaky"** *because* they provide

Complexity *Guarantees*

…where that complexity **directly results** from the **container implementation**

- This allows us:
  1. To **anticipate** *(plan for!)* expected behavior
  2. To **reason about** real-world *(actual!)* behavior

Container Complexity

`std::` containers are **necessarily "leaky"** *because* they provide

Complexity *Guarantees*

…where that complexity **directly results** from the **container implementation**

- This allows us:
  1. To **anticipate** *(plan for!)* expected behavior
  2. To **reason about** real-world *(actual!)* behavior

Necessary for ***Design***

Necessary to build systems that
- ***we control*** *and*
- ***we understand***

Container Complexity

`std::` containers are **necessarily "leaky"** *because* they provide

Complexity *Guarantees*

…where that complexity **directly results** from the **container implementation**

- This allows us:
  1. To **anticipate** *(plan for!)* expected behavior
  2. To **reason about** real-world *(actual!)* behavior

Necessary for ***Design***

Necessary to build systems that
- ***we control*** *and*
- ***we understand***

How do we leverage these guarantees?

We discuss issues in terms of **asymptotic** ***("Big-O")*** concepts *(and notation)*

The "Growth Rate": Big-O

- Recall ***Big-O***:
  - The ***"O"*** is "the ***Order*** of the function"
  - Is ***"the growth rate"***
  - Functions with **the same "growth rate"** have **the same Big-O notation**

"Big-O" Notation:
Describes the limiting behavior when the argument tends towards a particular value, or infinity

Also called:
- *Bachmann-Landau notation*
- *Asymptotic notation*

The "Growth Rate": Big-O

"Big-O" Notation:
Describes the limiting behavior when the argument tends towards a particular value, or infinity

- Recall ***Big-O***:
  - The ***"O"*** is "the ***Order*** of the function"
  - Is ***"the growth rate"***
  - Functions with **the same "growth rate"** have **the same Big-O notation**
- We use Big-O to **classify behavior for increased scale**:
  1. Algorithm **running time** *(i.e., "instructions complexity")*
  2. Container **space requirements**

Also called:
- *Bachmann-Landau notation*
- *Asymptotic notation*

The "Growth Rate": Big-O

"Big-O" Notation:
Describes the limiting behavior when the argument tends towards a particular value, or infinity

Also called:
- *Bachmann-Landau notation*
- *Asymptotic notation*

- Recall ***Big-O***:
  - The ***"O"*** is "the ***Order*** of the function"
  - Is ***"the growth rate"***
  - Functions with **the same "growth rate"** have **the same Big-O notation**
- We use Big-O to **classify behavior for increased scale**:
  1. Algorithm **running time** *(i.e., "instructions complexity")*
  2. Container **space requirements**
- **We select** containers, algorithms **based on Big-O** properties
  - *"Properties":*
    1. Intentionally **desired performance** characteristics *(i.e., "Functional")*
    2. Acceptably **tolerated limitations** *(i.e., "Non-Functional")*

The "Growth Rate": Big-O

"Big-O" Notation:
Describes the limiting behavior when the argument tends towards a particular value, or infinity

- Recall ***Big-O***:
  - The ***"O"*** is "the ***Order*** of the function"
  - Is ***"the growth rate"***
  - Functions with **the same "growth rate"** have **the same Big-O notation**

Also called:
- *Bachmann-Landau notation*
- *Asymptotic notation*

- We use Big-O to **classify behavior for increased scale**:
  1. Algorithm **running time** *(i.e., "instructions complexity")*
  2. Container **space requirements**

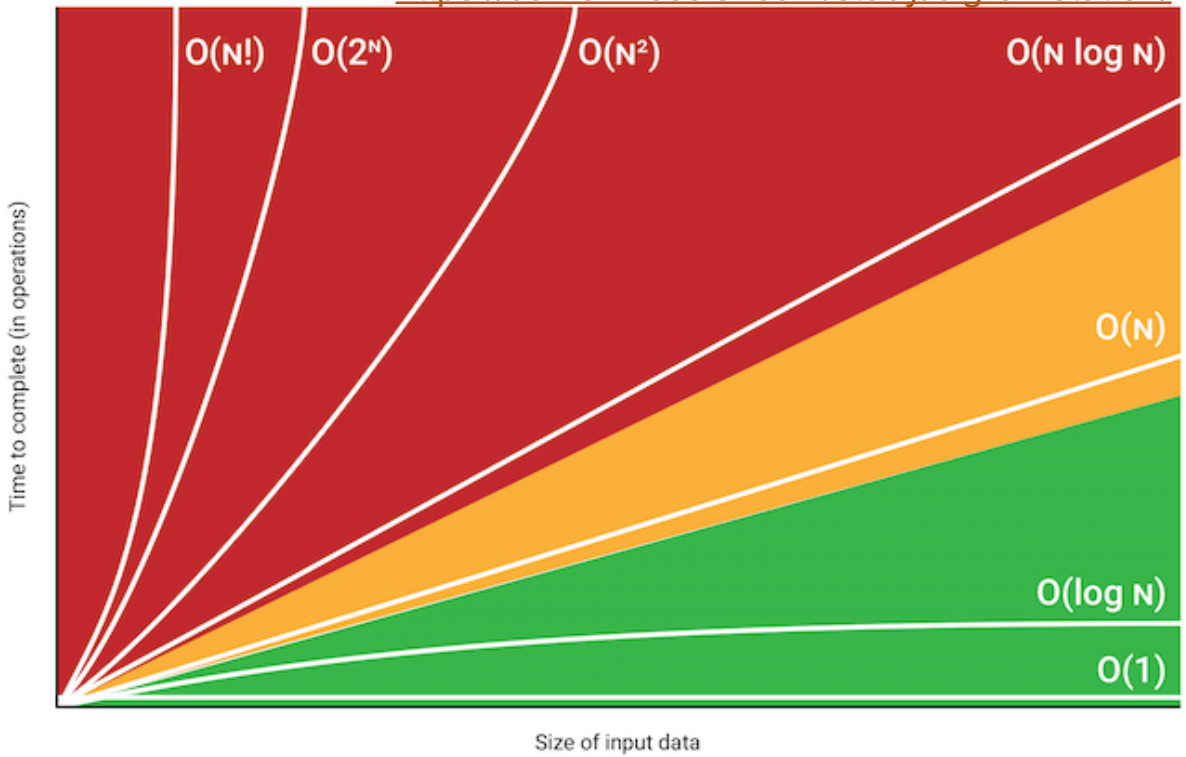
- **We select** containers, algorithms **based on Big-O** properties
  - *"Properties":*
    1. Intentionally **desired performance** characteristics *(i.e., "Functional")*
    2. Acceptably **tolerated limitations** *(i.e., "Non-Functional")*

Big-O is largely **irrelevant** at "small problems"

Big-O enables us to **contrast** how **efficiently** one {`container`|`algorithm`} solves **big problems** compared to another

Commonly Used Big-O
(Computer Science)

https://danielmiessler.com/study/big-o-notation/



Commonly Used Big-O
(Computer Science)

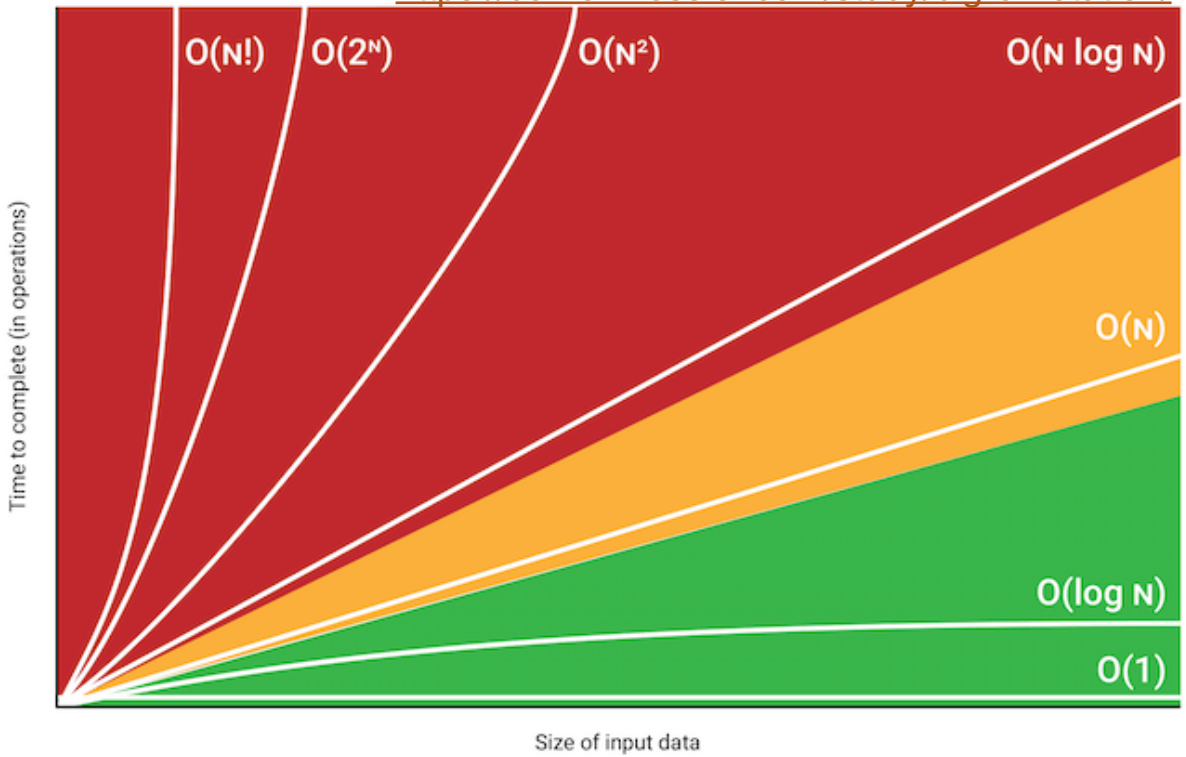
O(1) Item {`access`|`insertion`} takes the **same amount of time**, no matter how many items are in the container

"Constant Time"

- 1 item, 1 second
- 10 items, 1 second
- 100 items, 1 second

Optimal!

https://danielmiessler.com/study/big-o-notation/



Commonly Used Big-O
(Computer Science)

O(1) Item {`access` | `insertion`} takes the **same amount of time**, no matter how many items are in the container

"Constant Time"

- 1 item, 1 second
- 10 items, 1 second
- 100 items, 1 second

Optimal!

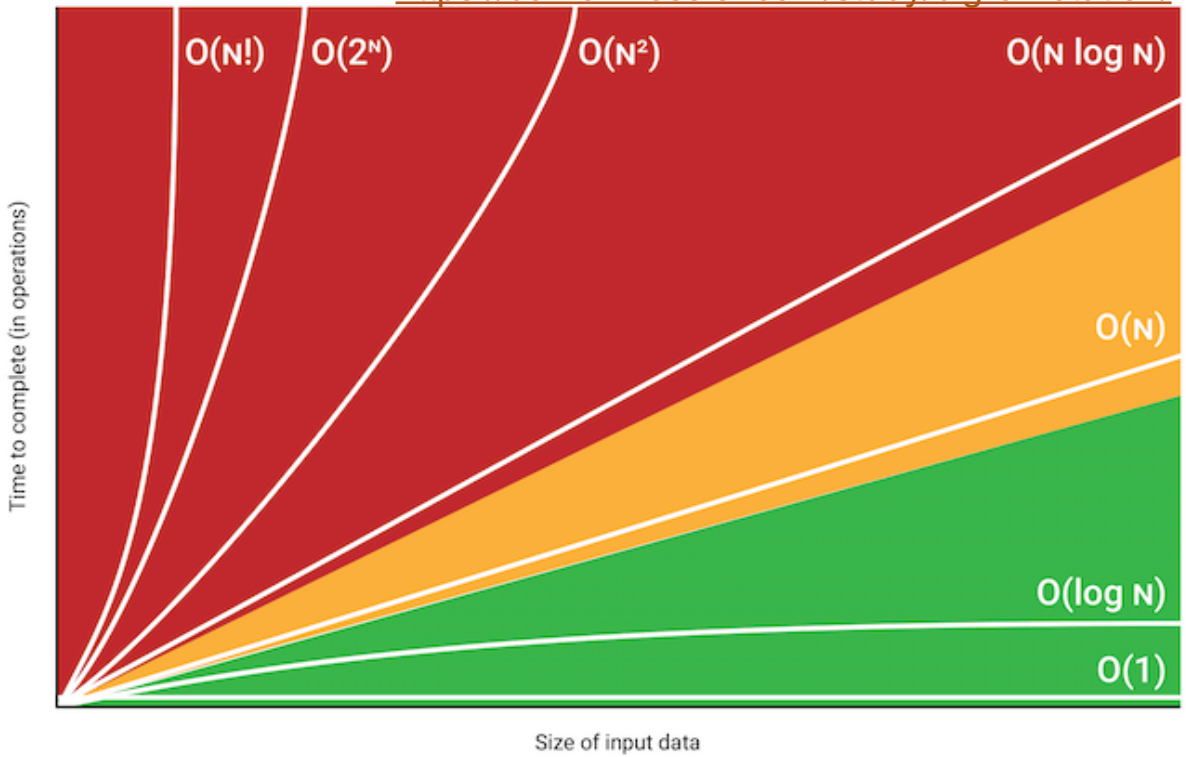
O(log n) Item {`access` | `insertion`} **cost barely increases** with exponential container size

"Logarithmic Time"

- 1 item, 1 second
- 10 items, 2 seconds
- 100 items, 3 seconds

almost Optimal!

https://danielmiessler.com/study/big-o-notation/



Commonly Used Big-O
(Computer Science)

O(1) Item {**access**|**insertion**} takes the **same amount of time**, no matter how many items are in the container

"Constant Time"

- 1 item, 1 second
- 10 items, 1 second
- 100 items, 1 second

Optimal!

O(log n) Item {**access**|**insertion**} **cost barely increases** with exponential container size

"Logarithmic Time"

- 1 item, 1 second
- 10 items, 2 seconds
- 100 items, 3 seconds

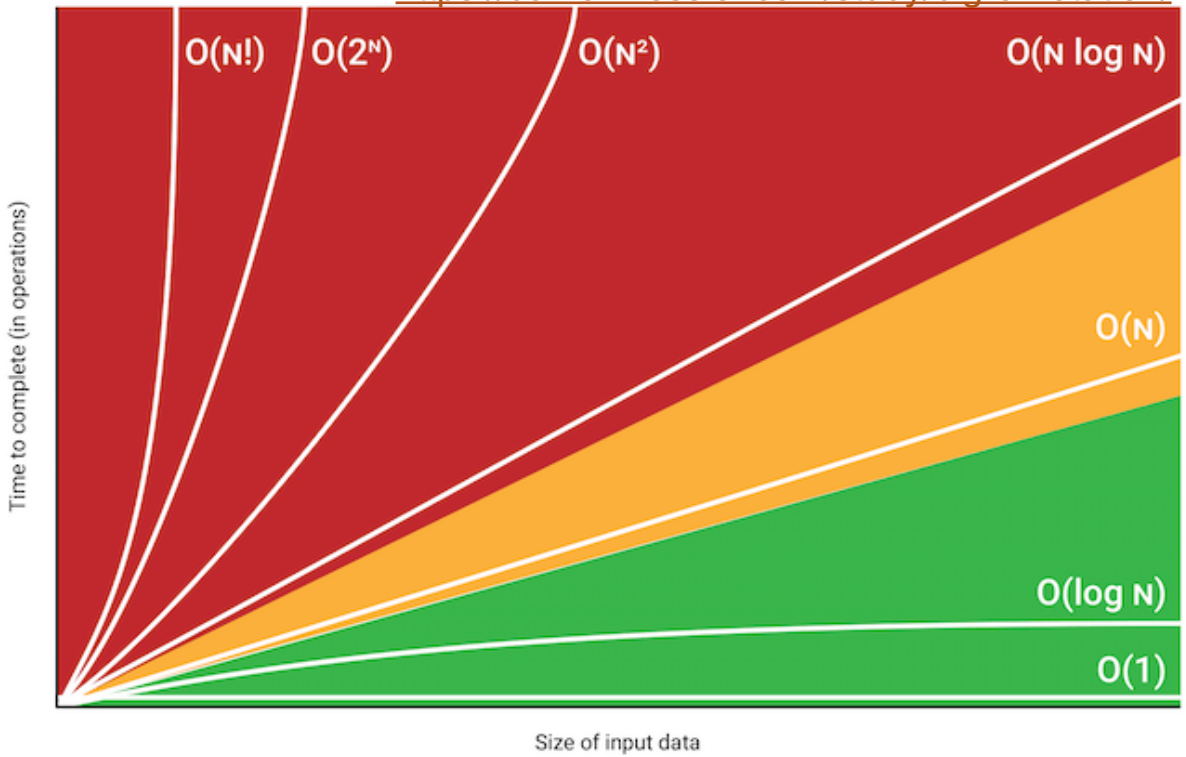
almost Optimal!

O(n) Item {**access**|**insertion**} cost **increases at the same pace** as container size

"Linear Time"

- 1 item, 1 second
- 10 items, 10 seconds
- 100 items, 100 seconds

https://danielmiessler.com/study/big-o-notation/



Commonly Used Big-O
(Computer Science)

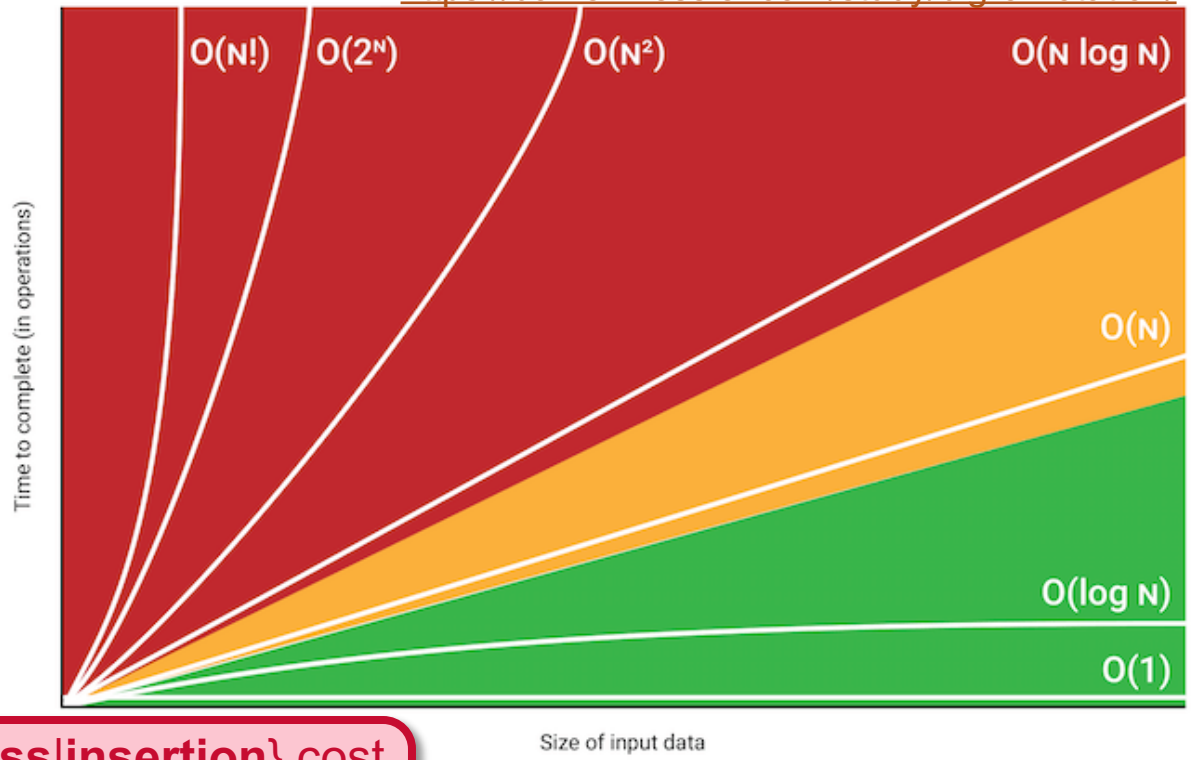
https://danielmiessler.com/study/big-o-notation/

O(1) Item {**access**|**insertion**} takes the **same amount of time**, no matter how many items are in the container

"Constant Time"

- 1 item, 1 second
- 10 items, 1 second
- 100 items, 1 second

Optimal!



O(log n) Item {**access**|**insertion**} **cost barely increases** with exponential container size

"Logarithmic Time"

- 1 item, 1 second
- 10 items, 2 seconds
- 100 items, 3 seconds

almost Optimal!

O(n) Item {**access**|**insertion**} cost **increases at the same pace** as container size

"Linear Time"

- 1 item, 1 second
- 10 items, 10 seconds
- 100 items, 100 seconds

O(n2) Item {**access**|**insertion**} cost **increases exponentially** with the container size

"Quadratic Time"

- 1 item, 1 second
- 10 items, 100 seconds
- 100 items, 10,000 seconds

RECALL:
*This is **quadratic***

Commonly Used Big-O
(Computer Science)

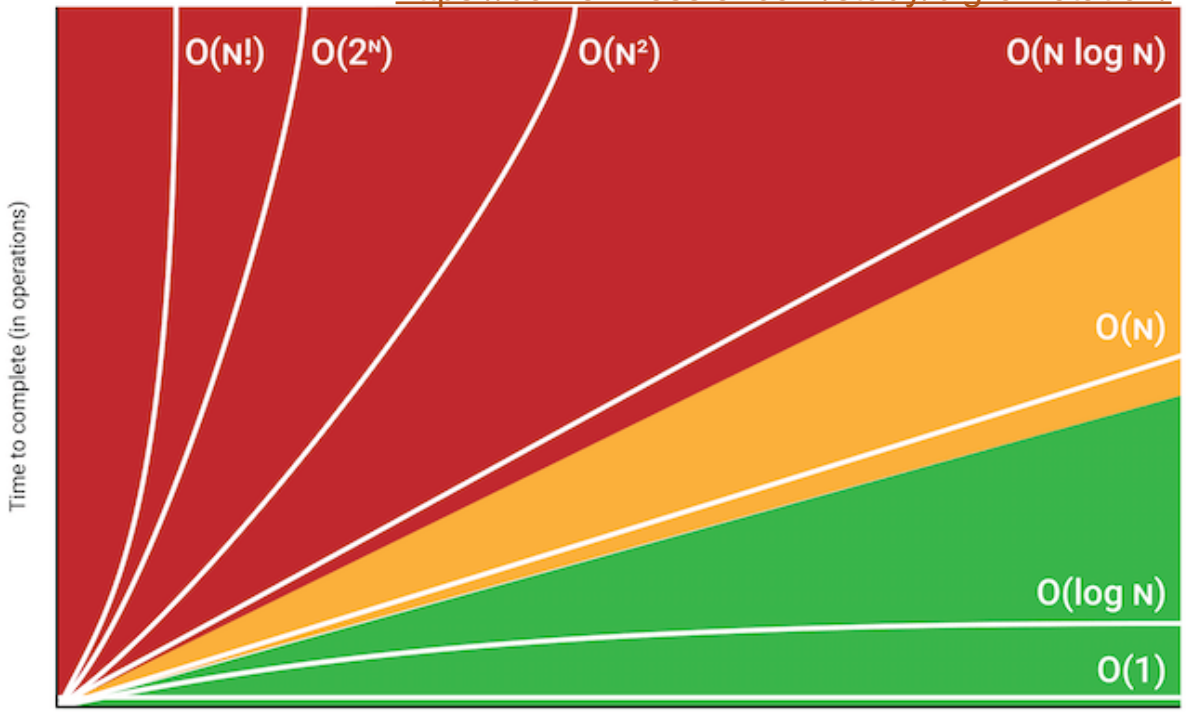
https://danielmiessler.com/study/big-o-notation/

O(1) Item {**access**|**insertion**} takes the **same amount of time**, no matter how many items are in the container

"Constant Time"

- 1 item, 1 second
- 10 items, 1 second
- 100 items, 1 second

Optimal!



O(log n) Item {**access**|**insertion**} **cost barely increases** with exponential container size

"Logarithmic Time"

- 1 item, 1 second
- 10 items, 2 seconds
- 100 items, 3 seconds

almost Optimal!

O(n) Item {**access**|**insertion**} cost **increases at the same pace** as container size

"Linear Time"

- 1 item, 1 second
- 10 items, 10 seconds
- 100 items, 100 seconds

O(n2) Item {**access**|**insertion**} cost **increases exponentially** with the container size

"Quadratic Time"

- 1 item, 1 second
- 10 items, 100 seconds
- 100 items, 10,000 seconds

*RECALL: This is **quadratic***

*RECALL: In the "algorithm world", **O(n!)** tends to mean **"unsolvable"** Example: Traveling Salesman*

O(n!) Item {**access**|**insertion**} cost **increases factorially** with the container size

"Factorial Time"

- 1 item, 1 second
- 10 items, 3,628,800 seconds
- 100 items, $9.332621544 \times 10^{157}$ seconds

Big-O Caveats

- Big-O usually **only specifies the "upper-bound"** behavior *(actual behavior may be better)*

Academic: Solves **theoretical** problems
Engineer: Solves **actual** problems

Big-O Caveats

- Big-O usually **only specifies the "upper-bound"** behavior *(actual behavior may be better)*
- For small "N", **Big-O is irrelevant** ← *Can be surpising!*
  - At "small-N", they all behave about the same
  - How big is "small-N"? *(Possibly very big!)*

Academic: Solves **theoretical** problems
Engineer: Solves **actual** problems

Big-O Caveats

- Big-O usually **only specifies the "upper-bound"** behavior *(actual behavior may be better)*
- For small "N", **Big-O is irrelevant** ← *Can be surpising!*
  - At "small-N", they all behave about the same
  - How big is "small-N"? *(Possibly very big!)*
- For a specific use, **Big-O can be misleading** ← *Can be surpising!*
  - The **"worst one" actually behaves the best** *(Why?)*
    - Specific algorithm may **space-shift and time-shift resource contention**
    - Specific data sets that make use of **out-of-order instruction processing**, or which perform **"training" of branch-prediction** in unexpected ways
    - Internal hardware or CPU optimizations **favoring some memory access patterns**

Academic: Solves **theoretical** problems
Engineer: Solves **actual** problems

Big-O Caveats

- Big-O usually **only specifies the "upper-bound"** behavior *(actual behavior may be better)*
- For small "N", **Big-O is irrelevant** ← *Can be surpising!*
  - At "small-N", they all behave about the same
  - How big is "small-N"? *(Possibly very big!)*
- For a specific use, **Big-O can be misleading** ← *Can be surpising!*
  - The **"worst one" actually behaves the best** *(Why?)*
    - Specific algorithm may **space-shift and time-shift resource contention**
    - Specific data sets that make use of **out-of-order instruction processing**, or which perform **"training" of branch-prediction** in unexpected ways
    - Internal hardware or CPU optimizations **favoring some memory access patterns**

Academic: Solves **theoretical** problems
Engineer: Solves **actual** problems

In practice, is **difficult to specify** precisely **when the Big-O constraint is satisfied**
(external context may cause actual behavior to be faster or slower than the theoretical or idealized behavior)

- Is it *really* constant-time to do a `vector`-copy? *(On all processors?)*
- Is it *really* constant-time to traverse a linked-list? *(Irrespective of paging and memory fragmentation?)*

Big-O Is A Guideline *(Not A Rule)*

- In practice, Big-O:
  - **May be generally true**, **but not specifically true** for your scenario *(and perhaps not ever true for your system)*
  - **May serve as a distraction** *(not important for your system, but preventing you from focusing on what is important)*
- **By necessity**, complexity specifications are an **oversimplification**
  - A full specification would describe variations in **all scenarios** *(example: "min" and "max" time for {`insert`|`access`} over range of increasing size)*
  - This data may be **unmanageable** for the user *(overloaded with irrelevant detail)*
  - A "complete" specification is **overly constraining** on the implementor *(specification would disallow optimizations)*

The C++ Standard is described through an abstract machine model. **Fully describing** asymptotic algorithm **complexity requires definition of constrained scenarios for real hardware** *(where such restrictions may not be true in practice for your scenario, even if you also used that same hardware)*

Invariant Management

Container Job #1:
Protect Its Invariants

Container Type Implies Additional Invariants

(examples):

- **Contiguous** containers: {`insert`} may require `realloc` *(and moving existing items)*
- **Associative** containers: {`insert`} of unique keys may still result in hash-collision *(unique keys must still be stored within the container)*

- Example Container Invariants:
  - {`insert`} If you put object in container, **it *must!* be there**
  - {`access`} If an object is in container, **it *must!* be accessible** somehow
  - {`erase`} If an object is removed from container, **it *must!* be gone**
- A container ***must*** choose to **sacrifice performance** to protect its invariants. *(Maintaining Invariants is Job#1)*

Invariant Management

Container Job #1:
Protect Its Invariants

Container Type Implies Additional Invariants
(examples):
- **Contiguous** containers: {`insert`} may require `realloc` *(and moving existing items)*
- **Associative** containers: {`insert`} of unique keys may still result in hash-collision *(unique keys must still be stored within the container)*

- Example Container Invariants:
  - {`insert`} If you put object in container, **it *must!* be there**
  - {`access`} If an object is in container, **it *must!* be accessible** somehow
  - {`erase`} If an object is removed from container, **it *must!* be gone**
- A container ***must*** choose to **sacrifice performance** to protect its invariants. *(Maintaining Invariants is Job#1)*
- Why?

If we cannot rely upon our invariants, ***chaos***:
We **cannot reason about anything** in our system

Declarative vs. Imperative

Logic Without Specified Control Flow

Declarative Programming

Declarative Programming *(def):*
A programming paradigm that expresses computation logic without describing its control flow

Programing Paradigm *(def):*
A style for building the structure and elements of computer programs

Declarative Programming

Declarative Programming *(def):*
A programming paradigm that expresses computation logic without describing its control flow

Programing Paradigm *(def):*
A style for building the structure and elements of computer programs

- ***Goal:*** **Minimize Non-Functional** requirements **by specifying *"what"*** must be accomplished **without specifying *"how"***
  1. **Intent** is more clear
  2. **Non-Functional** requirements are **avoided**
  3. **Greater optimization** opportunities exist *(**fewer constraints** are provided)*

Declarative Programming

Programing Paradigm *(def):*
A style for building the structure and elements of computer programs

Declarative Programming *(def):*
A programming paradigm that expresses computation logic without describing its control flow

- ***Goal:*** **Minimize Non-Functional** requirements **by specifying *"what"*** must be accomplished **without specifying *"how"***
  1. **Intent** is more clear
  2. **Non-Functional** requirements are **avoided**
  3. **Greater optimization** opportunities exist *(**fewer constraints** are provided)*

- **How Is This Done?**
  - Declarative languages **minimize or eliminate side-effects**
  - An **implementation** *(i.e., compiler or interpreter)* **decides "how"** a specification executes

Declarative Programming

Programing Paradigm *(def):*
A style for building the structure and elements of computer programs

Declarative Programming *(def):*
A programming paradigm that expresses computation logic without describing its control flow

- ***Goal:*** **Minimize Non-Functional** requirements **by specifying *"what"*** must be accomplished **without specifying *"how"***
  1. **Intent** is more clear
  2. **Non-Functional** requirements are **avoided**
  3. **Greater optimization** opportunities exist *(**fewer constraints** are provided)*

"Declarative" tends to be **applied** closer **to a specific domain**
*(where **domain rules** exist to **guide the selection** of "how")*

Implicit *or* ***Explicit*** *Domain Rules*

- **How Is This Done?**
  - Declarative languages **minimize or eliminate side-effects**
  - An **implementation** *(i.e., compiler or interpreter)* **decides "how"** a specification executes

Declarative Languages

"A Very High-Level Taste" of Declarative Languages

- **Declarative *and* Imperative** are largely considered ***opposites***
- **Functional programming** style tends to **look declarative**

  (so declarative language descriptions tend to share functional concepts)
  - **Functional programming** is **not limited to declarative** programming.
- **Declarative Languages**:
  - **Express logical "steps"**

    (through nested function call order, i.e., "sub-expressions")
  - **Don't have looping control structures**

    (because due to immutability, the loop condition would never change)
  - **Don't express control-flow other than nested function order**

    (because due to immutability, evaluation order of sub-expressions does not change the result)

Declarative Languages
(examples):
Erlang, Prolog, QML, SQL, XSLT

Interesting discussion, credit to:
Shelby Moore III


https://stackoverflow.com/questions/602444/functional-declarative-and-imperative-programming/8357604#8357604

Shifting Gears: Declarative vs. Imperative

- Until now, we contrasted “functional” and “non-functional” requirements:

Functional: The **desired behavior** *(i.e., “Requirement”)*

Non-Functional: The **result of a leaky abstraction** *(i.e., “Technical Requirement”)*

Shifting Gears: Declarative vs. Imperative

- Until now, we contrasted “functional” and “non-functional” requirements:

  Functional: The **desired behavior** *(i.e., “Requirement”)*

  Non-Functional: The **result of a leaky abstraction** *(i.e., “Technical Requirement”)*

- Now, we contrast “imperative” and “declarative”:

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

Shifting Gears: Declarative vs. Imperative

- Until now, we contrasted "functional" and "non-functional" requirements:

Functional: The **desired behavior** *(i.e., "Requirement")*

Non-Functional: The **result of a leaky abstraction** *(i.e., "Technical Requirement")*

- Now, we contrast "imperative" and "declarative":

1 **Declarative:**
You write ***what*** you want, ***not how*** to get it

2 **Imperative:**
You **specify control flow** to achieve a result

- Most real-world software systems are a **mix of multiple paradigms**
  - *(i.e., "Best Tool For The Job")*

HINT: You are **already doing "Declarative"** in some parts of your system

Container Interfaces

- For `std::` containers,
  - What is an example of an “**imperative” interface**?

1 **Declarative:**
You write ***what*** you want, ***not how*** to get it

2 **Imperative:**
You **specify control flow** to achieve a result

Container Interfaces

- For `std::` containers,
  - What is an example of an "**imperative" interface**?
    - ***Iterators!*** *(Hint: Range-based* `for` *loop uses iterators)*

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

Hard to get more "imperative" than using an iterator!

Container Interfaces

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

- For **`std::`** containers,
  - What is an example of an "**imperative" interface**?
    - ***Iterators!*** *(Hint: Range-based* ***`for`*** *loop uses iterators)*

Hard to get more "imperative" than using an iterator!

```
int main() {
  std::unordered_map<std::string, std::string> u = {
   {"RED","#FF0000"},{"GREEN","#00FF00"},{"BLUE","#0000FF"} };
  for( const auto& n : u ) {
    std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
  }
  return 0;
}
```

(Example from):
https://en.cppreference.com/w/cpp/container/unordered_map

You own the Control Flow!
Do whatever you want!

- Control flow is **entirely specified by you**.

The Joy Of Owning Control Flow

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

```
std::vector<SdpVideoFormat>
StereoDecoderFactory::GetSupportedFormats() const {
 std::vector<SdpVideoFormat> formats = ....;
 for (const auto& format : formats) {
   if (cricket::CodecNamesEq(....)) {
     ....
     formats.push_back(stereo_format);
   }
 }
 return formats;
}
```

The Joy Of Owning Control Flow

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

```
std::vector<SdpVideoFormat>
StereoDecoderFactory::GetSupportedFormats() const {
 std::vector<SdpVideoFormat> formats = ....;
 for (const auto& format : formats) {
   if (cricket::CodecNamesEq(....)) {
     ....
     formats.push_back(stereo_format);
   }
 }
 return formats;
}
```

PVS-Studio warning: V789 CWE-672 Iterators for the 'formats' container, used in the range-based for loop, become invalid upon the call of the 'push_back' function.

The Joy Of Owning Control Flow

GGribkov March 20, 2019 at 05:10 PM

Top 10 bugs of C++ projects found in 2018

PVS-Studio corporate blog, C++, Open source, Programming

- 8th Place: Chromium *(WebRTC Library)*

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

```
std::vector<SdpVideoFormat>
StereoDecoderFactory::GetSupportedFormats() const {
 std::vector<SdpVideoFormat> formats = ....;
 for (const auto& format : formats) {
   if (cricket::CodecNamesEq(....)) {
     ....
     formats.push_back(stereo_format);
   }
 }
 return formats;
}
```

PVS-Studio warning: V789 CWE-672 Iterators for the 'formats' container, used in the range-based for loop, become invalid upon the call of the 'push_back' function.

"The error is that the **size of the *formats* vector varies** within the range-based for loop. **Range-based loops are based on iterators**, that's why changing of the container size inside of such loops might result in invalidation of these iterators." *(emphasis added)*

https://habr.com/en/company/pvs-studio/blog/444568/

The Joy Of Owning Control Flow *(continued)*

GGribkov March 20, 2019 at 05:10 PM

Top 10 bugs of C++ projects found in 2018

PVS-Studio corporate blog, C++, Open source, Programming

- 8th Place: Chromium *(WebRTC Library)*

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

```
std::vector<SdpVideoFormat> StereoDecoderFactory::GetSupportedFormats() const {
 std::vector<SdpVideoFormat> formats = ....;
 for (const auto& format : formats) {
   if (cricket::CodecNamesEq(....)) {
     ....
     formats.push_back(stereo_format);
   }
 }
 return formats;
}
```

Capture ***begin()*** *and* ***end()*** *(at beginning of loop)*

```
for (auto format = begin(formats), __end = end(formats);
          format != __end; ++format) {
  if (cricket::CodecNamesEq(....)) {
     ....
     formats.push_back(stereo_format);
  }
}
```

format *iterator invalidated (perhaps* ***realloc****); end() might change (from* ***realloc****)*

"This error persists, if rewrite the loop with an explicit usage of iterators."

To avoid such errors, follow the rule: **never change a container size inside a loop with conditions bound to this container**. It **also relates to range-based loops and loops using iterators**. You're welcome to read this discussion on StackOverflow that covers the topic of operations causing invalidation of iterators.

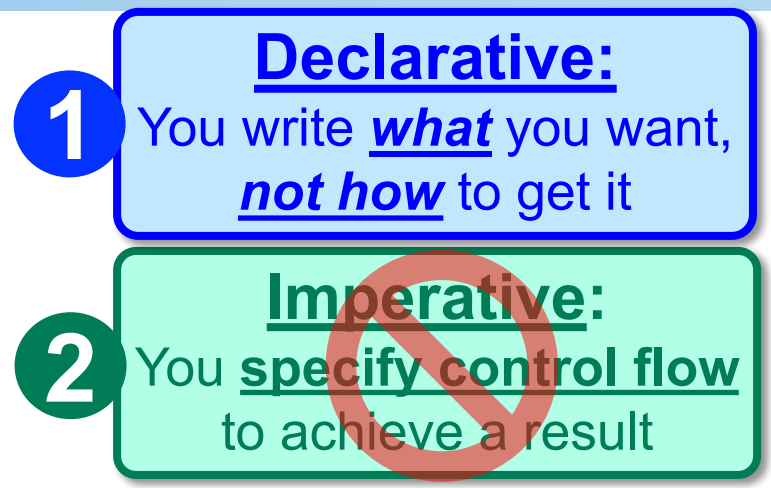
(emphasis added)

https://habr.com/en/company/pvs-studio/blog/444568/

https://stackoverflow.com/questions/6438086/iterator-invalidation-rules

Container Interfaces

- For `std::` containers,
  - What is an example of a “**declarative” interface**?



Container Interfaces

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

- For `std::` containers,
  - What is an example of a "**declarative" interface**?

```
<some-container>::size() // return the number of elements
```

- "**How" is not specified**:
  - *Does it return a cached value?*
  - *Does it access a property from a nested container?*
  - *Does it iterate all nodes in a chain?*
  - *Does it accumulate counts from iterating allocation slabs?*
  - *Does it …?*

None of your business!

Case Study: `std::forward_list`

- `std::forward_list` *(since C++11)*
  - Supports **fast insertion and removal**
  - Random access **not supported**
  - Implementation: **singly-linked list**
  - **More space efficient** than `std::list` *(a doubly-linked list)*
- What are the requirements?

Case Study: `std::forward_list`

- `std::forward_list` *(since C++11)*
  - Supports **fast insertion and removal**
  - Random access **not supported**
  - Implementation: **singly-linked list**
  - **More space efficient** than `std::list` *(a doubly-linked list)*

Sorting In `std::`

`std::sort`	*requires random-access iterator*
`std::stable_sort`	*requires random-access iterator*
`std::partial_sort`	*requires random-access iterator*
`std::list::sort`	*sort list elements*
`std::forward_list::sort`	*sort list elements*
`std::pmr::list::sort`	*sort list elements*
`std::pmr::forward_list::sort`	*sort list elements*
`std::qsort`	*sort on `void*` (no iterator required)*

- What are the requirements?

Functional:
- Fast {`insert` | `erase`}
- Forward-iteration only
- More space efficient than `std::list`

Non-Functional: *(technical constraint)*
- Not supported:
  - Random-access, Reverse-access
- Implementation: singly-linked list

*Implementation is **unfriendly** to many generic algorithms! (example: `sort()`)*

Declarative Container Interfaces

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

- Declarative **Manipulation**:
  - *(Example from):* https://en.cppreference.com/w/cpp/container/forward_list/reverse

```
int main() {
 std::forward_list<int> list = { 8,7,5,9,0,1,3,2,6,4 };
 std::cout << "before: " << list << "\n";
 list.sort();
 std::cout << "ascending: " << list << "\n";


}
```

HOW *is this done?*

Output

```
before:     8 7 5 9 0 1 3 2 6 4
ascending:  0 1 2 3 4 5 6 7 8 9
```

Declarative Container Interfaces

1. **Declarative:**
   You write ***what*** you want, ***not how*** to get it

2. **Imperative:**
   You **specify control flow** to achieve a result

- Declarative **Manipulation**:
  - *(Example from):*
    https://en.cppreference.com/w/cpp/container/forward_list/reverse

```
int main() {
 std::forward_list<int> list = { 8,7,5,9,0,1,3,2,6,4 };
 std::cout << "before: " << list << "\n";
 list.sort();
 std::cout << "ascending: " << list << "\n";
 list.reverse();
 std::cout << "descending: " << list << "\n";
}
```

HOW *is this done?*

HOW *is this done?*

Output

```
before:      8 7 5 9 0 1 3 2 6 4
ascending:   0 1 2 3 4 5 6 7 8 9
descending:  9 8 7 6 5 4 3 2 1 0
```

Declarative Container Interfaces

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

- Declarative **Manipulation**:
  - *(Example from):* https://en.cppreference.com/w/cpp/container/forward_list/reverse

```
int main() {
 std::forward_list<int> list = { 8,7,5,9,0,1,3,2,6,4 };
 std::cout << "before: " << list << "\n";
 list.sort();
 std::cout << "ascending: " << list << "\n";
 list.reverse();
 std::cout << "descending: " << list << "\n";
}
```

HOW *is this done?*

HOW *is this done?*

(Example)

Declarative Interface:

```
std::forward_list::reverse()
std::forward_list::sort()
```

- **"How" is not specified**
  (it is none of your business)

Output

```
before:      8 7 5 9 0 1 3 2 6 4
ascending:   0 1 2 3 4 5 6 7 8 9
descending:  9 8 7 6 5 4 3 2 1 0
```

Declarative Container Interfaces

- Declarative **Manipulation** *(continued):*
  - *(Example from):* https://en.cppreference.com/w/cpp/container/forward_list/unique

```
int main() {
  std::forward_list<int> x = {1, 2, 2, 3, 3, 2, 1, 1, 2};
  std::cout << "contents before:";
  for (auto val : x)
    std::cout << ' ' << val;
  std::cout << '\n';
  x.unique();
  std::cout << "contents after unique():";
  for (auto val : x)
    std::cout << ' ' << val;
  std::cout << '\n';
  return 0;
}
```

***HOW** is this done?*

  - **"How" is not specified**

    (it is none of your business)

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You **specify control flow** to achieve a result

(Example)
Declarative Interface:

```
std::forward_list::reverse()
std::forward_list::sort()
std::forward_list::unique()
```

Output

```
contents before: 1 2 2 3 3 2 1 1 2
contents after unique(): 1 2 3 2 1 2
```

Declarative Container Interfaces

- Declarative **Manipulation** *(continued):*
  - *(Example from):* https://en.cppreference.com/w/cpp/container/forward_list/remove

```
int main() {
  std::forward_list<int> l = { 1,100,2,3,10,1,11,-1,12 };
  l.remove(1); // remove both elements equal to 1
  l.remove_if([](int n){ return n > 10; }); // remove all elements greater than 10
  for (int n : l) {
    std::cout << n << ' '; }
  std::cout << '\n';
}
```

HOW is this done?

HOW is this done? (Lambda is a predicate for "what" items to remove, not "how" to remove)

Output

```
2 3 10 -1
```

- "How" is not specified
  (it is none of your business)

1 Declarative:
You write ***what*** you want, ***not how*** to get it

2 Imperative:
You specify control flow to achieve a result

(Example)
Declarative Interface:

```
std::forward_list::remove()
std::forward_list::remove_if()
std::forward_list::reverse()
std::forward_list::sort()
std::forward_list::unique()
```

Contrasting Declarative vs. Imperative

	Declarative	Imperative
Control	You **give up control** *(over "How")*	You **retain fine-grained control** *(over execution flow)*
Code Size	**Less** code!	**More** code!
Iteration Details	Iteration details are **"implied" or hidden**	Iteration details are **exposed**
Ease-of-use	Very **easy to use**! *(Simple expression!)*	You create: (1) iteration **control structure**; and (2) and **iteration body**
Edge cases	**Fewer** apparent edge cases	**You manage all edge cases and side-effects** *(between your iteration control structure and iteration body operating on localized context)*
Localized Context	Localized context is **ignored**, **or explicitly injected**	Localized context is **implicitly coupled** as part of the algorithmic scope
Composition	Easy to **compose simple** expressions	Can **compose** many more types of **advanced expressions** with rich control of side-effects by coupling to localized context
Limitations	Is **limited**	Is *(relatively)* **unlimited**

Example:

```
std::forward_list::remove_if(<your-predicate>)
```

Contrasting Declarative vs. Imperative

- **Summary Observations** *(contrasting Declarative vs. Imperative):*

Declarative:
You give up control *(over "How")* for simplified expression of common use cases

*"The **80** in the 80/20 rule"*

***Why Not**?*
*You might need greater control over "How", like to (efficiently) trigger **multiple side-effects** or compute **multiple by-products** from a **single** loop iteration*

Imperative:
You retain fine-grained control-flow to efficiently couple with localized context

*"The **20** in the 80/20 rule"*

***Why Not**?*
*You could instead write **simpler expressions** with **less code** that **avoids** many types of **errors and edge cases***

Container Interface Mechanisms

The Public Interface

What Is A “Minimal” Container Interface?

- A **Container holds objects**. Therefore, all containers must minimally provide some form of:
  - {`insert`} …object is **added** to the container
  - {`access`} …object in a container is **accessed** *(implicitly or explicitly)*
  - {`erase`} …object is **removed** *(may require emptying entire container)*

What Is A “Minimal” Container Interface?

- A **Container holds objects**. Therefore, all containers must minimally provide some form of:
  - {`insert`} …object is **added** to the container
  - {`access`} …object in a container is **accessed** *(implicitly or explicitly)*
  - {`erase`} …object is **removed** *(may require emptying entire container)*

Q: What is a good example of “accessing” members within containers?

What Is A "Minimal" Container Interface?

- A **Container holds objects**. Therefore, all containers must minimally provide some form of:
  - {`insert`} …object is **added** to the container
  - {`access`} …object in a container is **accessed** *(implicitly or explicitly)*
  - {`erase`} …object is **removed** *(may require emptying entire container)*

Q: What is a good example of "accessing" members within containers?

A: Algorithms!

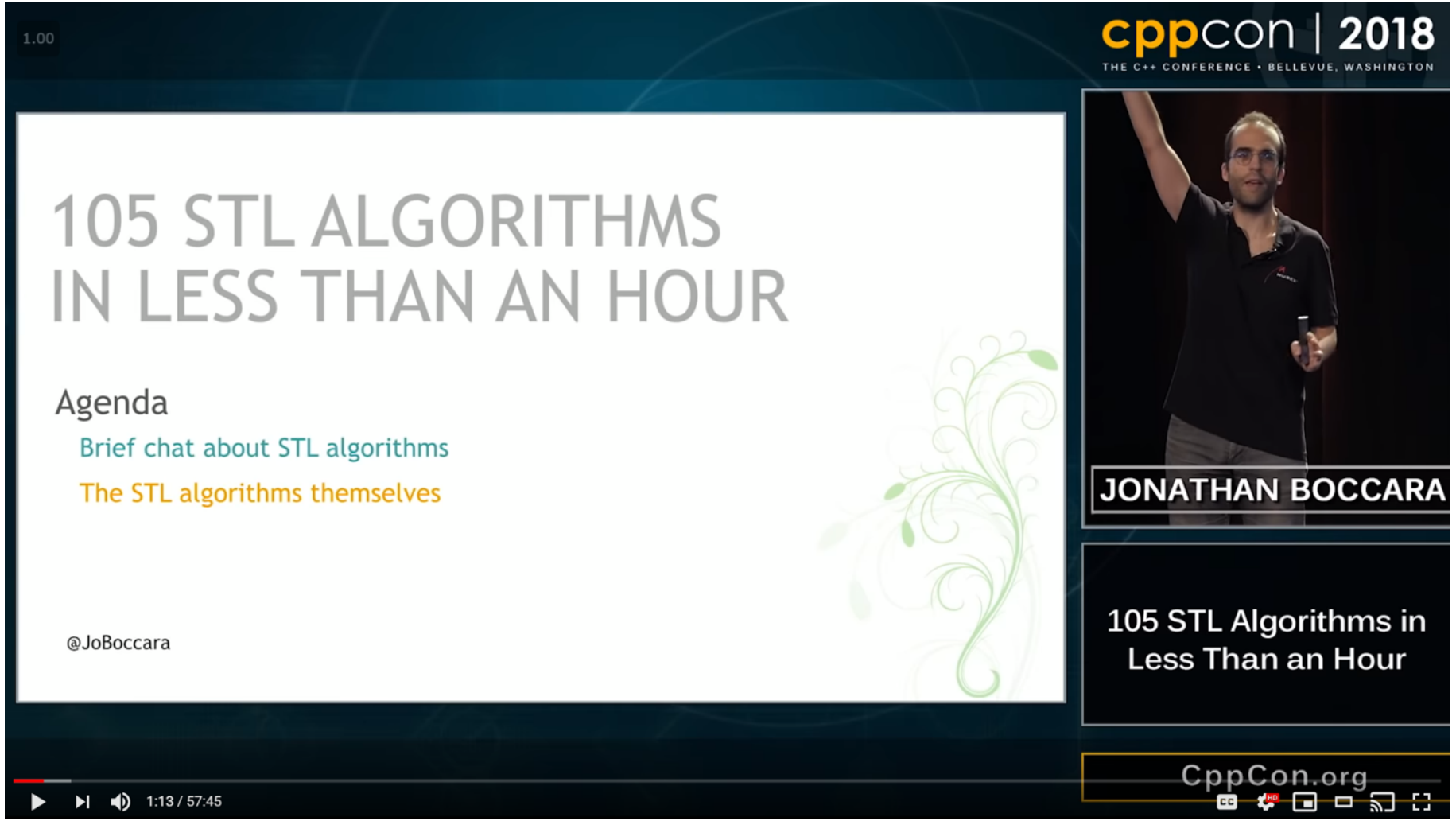
C++ Standard Algorithms

CppCon 2018: Jonathan Boccara **“105 STL Algorithms in Less Than an Hour”**

Watch It.

Know It.

Love It.



https://www.youtube.com/watch?v=2olsGf6JIkU

Container Interface Mechanisms

Container Interface Mechanism:
The provided tool intended for accessing container contents

- Two approaches exist for accessing container elements:

Container Interface Mechanisms

Container Interface Mechanism:
The provided tool intended for accessing container contents

- Two approaches exist for accessing container elements:

2 **Iteration** *(imperative)*

- **Iterate your items**
- *Examples:*
  - Index into sequential containers
  - Iterators, `std::` algorithms
  - *(syntactic sugar):* Range-based `for`, ranges, etc.

Recall: `std::` *algorithms operate using* ***iterators***

Container Interface Mechanisms

Container Interface Mechanism:
The provided tool intended for accessing container contents

- Two approaches exist for accessing container elements:

1 Property Extraction *(declarative)*

- **How** is the property provide?
  (Is none of your business!)
- *Examples:*

```
<some-container>::size()       // return element count
<some-container>::empty()      // return true if no elements

<some-container>::remove_if()  // drop items matching predicate
<some-container>::reverse()    // reverse element order
<some-container>::sort()       // order all elements
<some-container>::unique()     // drop all duplicates
```

*"Property": Returns **count removed***

*Triggers **side-effects**!*

2 Iteration *(imperative)*

- **Iterate your items**
- *Examples:*
  - Index into sequential containers
  - Iterators, `std::` algorithms
  - *(syntactic sugar):* Range-based `for`, ranges, etc.

*Recall: `std::` algorithms operate using **iterators***

Property Extraction Is *"Information Leakage"*

"Property" Extraction: *Deriving state* from container contents

- A "**Property Extraction" interface**:

1. Might be `const` *(examples):*
   - Read of cached value
   - Read-only iteration of contents *(i.e., no side-effects)*

2. Might be non-`const` *(examples):*
   - Destructive "read"
   - Property computed as by-product of executing a side-effect

Property Extraction Is *"Information Leakage"*

"Property" Extraction: *Deriving state* from container contents

- A "**Property Extraction" interface**:

1. Might be **`const`** *(examples):*
   - Read of cached value
   - Read-only iteration of contents *(i.e., no side-effects)*

2. Might be non-**`const`** *(examples):*
   - Destructive "read"
   - Property computed as by-product of executing a side-effect

- The ***"side-effect"* might imply** the ***"property"***

```
std::forward_list<int> my_list = { /*…items…*/ };
my_list.unique();
... // By here, we "know" that "my_list" has unique elements
    // (is local property that we can "reason about")
```

Information "leakage" *into local context (from computing side-effect)*

Information Leakage *Through* ***Side Effect***

Information Leakage *(into the local context)* **occurs when**
a side-effect provides a local property that we can *"reason about"*

Information Leakage *Through* ***Side Effect***

Information Leakage *(into the local context)* **occurs when a side-effect provides a local property** that we can *"reason about"*

Information "leakage" *(provides local property that we can "reason about")*

```
std::forward_list<int> my_list = { /*…items…*/ };
my_list.unique();
... // By here, we "know" that "my_list" has unique elements
```

Property

Information Leakage *Through* ***Side Effect***

Information Leakage *(into the local context)* **occurs when a side-effect provides a local property** that we can *"reason about"*

Information "leakage" *(provides local property that we can "reason about")*

```
std::forward_list<int> my_list = { /*…items…*/ };
my_list.unique();
... // By here, we "know" that "my_list" has unique elements

my_list.remove_if([](int n){ return n > 10; });
... // By here, we "know" that "my_list" has no value greater than "10"
```

Property

Information Leakage *Through* ***Side Effect***

Information Leakage *(into the local context)* **occurs when a side-effect provides a local property** that we can *"reason about"*

Information "leakage" *(provides local property that we can "reason about")*

```
std::forward_list<int> my_list = { /*…items…*/ };
my_list.unique();
... // By here, we "know" that "my_list" has unique elements

my_list.remove_if([](int n){ return n > 10; });
... // By here, we "know" that "my_list" has no value greater than "10"

my_list.sort();
... // By here, we "know" that "my_list" is sorted
```

Property

“Property” vs. “Attribute”

“Property”: A stateful “snapshot” in time

```
std::vector<Dog> my_vec;
... = my_vec.size();   // property (can change over time)
```

“Property” vs. “Attribute”

You **might not care** *(much)* about whether something is a “Property” or “Attribute”
(might be academic for your use case)

“Property”: A stateful “snapshot” in time

```
std::vector<Dog> my_vec;
... = my_vec.size();   // property (can change over time)
```

“Attribute”: An *(inherent)* characteristic

```
std::array<Dog,42> my_array;
... = my_array.size(); // attribute (inherent to type)
```

“Property” vs. “Attribute”

You **might not care** *(much)* about whether something is a “Property” or “Attribute” *(might be academic for your use case)*

“Property”: A stateful “snapshot” in time

```
std::vector<Dog> my_vec;
... = my_vec.size();   // property (can change over time)
```

“Attribute”: An *(inherent)* characteristic

```
std::array<Dog,42> my_array;
... = my_array.size(); // attribute (inherent to type)
```

“Attribute” vs. “Property” can be a “gray area”

```
struct MyDogs {
  using DogVec = std::vector<Dog>;
  using DogList = std::forward_list<Dog>;
  std::variant<DogVec, DogList> impl_;
  bool is_contiguous(void) const {
    return std::holds_alternative<DogVec>(impl_);
  }
};
```

```
void foo(void) {
  MyDogs my_dogs;
  // property (can change over time)
  if(my_dogs.is_contiguous()) {
    // ...
  }
}
```

Storage type may change at runtime

Which Mechanism?

- Your interface mechanism is…

Declarative:
Provides state or side-effects, but hides detail of "How"

Imperative:
Enables user-directed control flow over "How"

Defining A Container Interface

Separating “Interface” from “Implementation”

Perl is a language for getting your job done.

> Of course, if your job is programming, you can get your job done with any "complete" computer language, theoretically speaking. But we know from experience that computer languages differ not so much in what they make possible, but in what they make easy. At one extreme, the so-called "fourth generation languages" make it easy to do some things, but nearly impossible to do other things. At the other extreme, certain well known, "industrial-strength" languages make it equally difficult to do almost everything.
>
> Perl is different. In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible.

Larry Wall

Programming Perl, 2nd Edition (1996) by Larry Wall, Tom Christiansen, and Randal Schwartz

Declarative paradigm

Perl is a language for getting your job done.

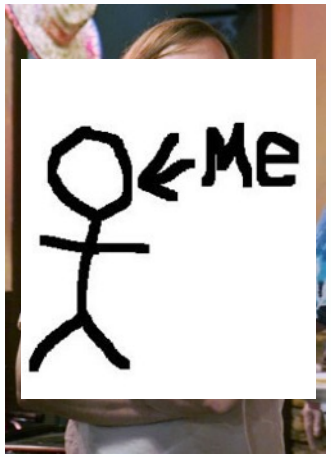
> Of course, if your job is programming, you can get your job done with any "complete" computer language, theoretically speaking. But we know from experience that computer languages differ not so much in what they make possible, but in what they make easy. At one extreme, the so-called "fourth generation languages" make it easy to do some things, but nearly impossible to do other things. At the other extreme, certain well known, "industrial strength" languages make it equally difficult to do almost everything.
>
> Perl is different. In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible.
>
> Larry Wall

interface paradigm

interface paradigms

functional monadic paradigms

imperative paradigms



Declarative

Declarative

charley

Programming Perl, 2nd Edition (1996) by Larry Wall, Tom Christiansen, and Randal Schwartz

Easy things should be easy,

and hard things should be possible.

Larry Wall

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:









};
```

*Storage for **Dog** or **Dog***

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

What If:
Create a CRTP container that abstracts storage for "**object value**" *or* "**address value**"

*std::**remove_pointer_t***
(Since C++17)

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native);




};
```

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

*Storage for **Dog** or **Dog***

*std::**remove_pointer_t** (Since C++17)*

What If:
Create a CRTP container that abstracts storage for **"object value"** *or* **"address value"**

***add** "natively-stored" item*

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native);

    TYPE_ITEM_CONST_PTR getItem_as_ptr(std::size_t index) const;
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index);




};
```

*Storage for **Dog** or **Dog***

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

What If:
Create a CRTP container that abstracts storage for **"object value"** *or* **"address value"**

std::remove_pointer_t *(Since C++17)*

add *"natively-stored" item*

get *"natively-stored" item (as ptr)*

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native);

    TYPE_ITEM_CONST_PTR getItem_as_ptr(std::size_t index) const;
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index);

    const TYPE_ITEM_NATIVE& getItem_native(std::size_t index) const;
    TYPE_ITEM_NATIVE& getItem_native(std::size_t index);




};
```

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

*Storage for **Dog** or **Dog***

*std::**remove_pointer_t***
(Since C++17)

What If:
Create a CRTP container that abstracts storage for **"object value"** *or* **"address value"**

***add** "natively-stored" item*

***get** "natively-stored" item (as ptr)*

***get** "natively-stored" item (by ref)*

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native);

    TYPE_ITEM_CONST_PTR getItem_as_ptr(std::size_t index) const;
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index);

    const TYPE_ITEM_NATIVE& getItem_native(std::size_t index) const;
    TYPE_ITEM_NATIVE& getItem_native(std::size_t index);

    const VorPv_native& get_VorPv_native_ref(void) const;
    VorPv_native& get_VorPv_native_ref(void);


};
```

*Storage for **Dog** or **Dog***

*std::**remove_pointer_t***
(Since C++17)

What If:
Create a CRTP container that abstracts storage for **"object value"** *or* **"address value"**

***add** "natively-stored" item*

***get** "natively-stored" item (as ptr)*

***get** "natively-stored" item (by ref)*

***get** "container implementation" (by ref)*

Case Study: Abstracting A Container

*"C++ is a **value semantic** language"*

In C++, a **Container** is a **collection of values**, one of:

1. **Data object** value
2. **Address** value

Remember?

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using VorPv_native          = VorPv_native_;
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
    using TYPE_ITEM             = std::remove_pointer_t<TYPE_ITEM_NATIVE>;
    using TYPE_ITEM_PTR         = TYPE_ITEM *;
    using TYPE_ITEM_CONST_PTR   = TYPE_ITEM const*;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native);

    TYPE_ITEM_CONST_PTR getItem_as_ptr(std::size_t index) const;
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index);

    const TYPE_ITEM_NATIVE& getItem_native(std::size_t index) const;
    TYPE_ITEM_NATIVE& getItem_native(std::size_t index);

    const VorPv_native& get_VorPv_native_ref(void) const;
    VorPv_native& get_VorPv_native_ref(void);

    std::size_t getNumItems(void) const;
};
```

*Storage for **Dog** or **Dog***

*std::**remove_pointer_t***
(Since C++17)

What If:
Create a CRTP container that abstracts storage for **"object value"** *or* **"address value"**

***add** "natively-stored" item*

***get** "natively-stored" item (as ptr)*

***get** "natively-stored" item (by ref)*

***get** "container implementation" (by ref)*

***get** "number-of-items"*

Case Study: CRTP Hierarchy

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
    ...
    std::size_t getNumItems(void) const;
};
```

Goal: Generic logic

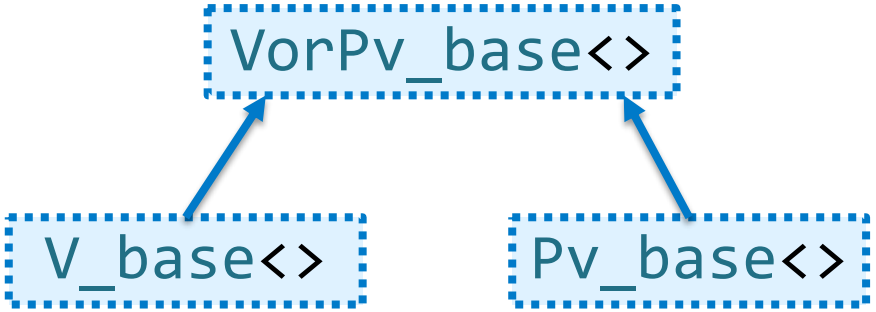
Generic
(reusable across domains)

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

Goal: Empty

insert storage spec

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```



Case Study: CRTP Hierarchy

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
    ...
    std::size_t getNumItems(void) const;
};
```

Goal: Generic logic

Generic
(reusable across domains)

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

Goal: Empty

insert storage spec

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

```
template<class DERIVED, class TYPE_VorPv_base>
class DogVorPv_base : public TYPE_VorPv_base {
};
```

Goal: Domain logic

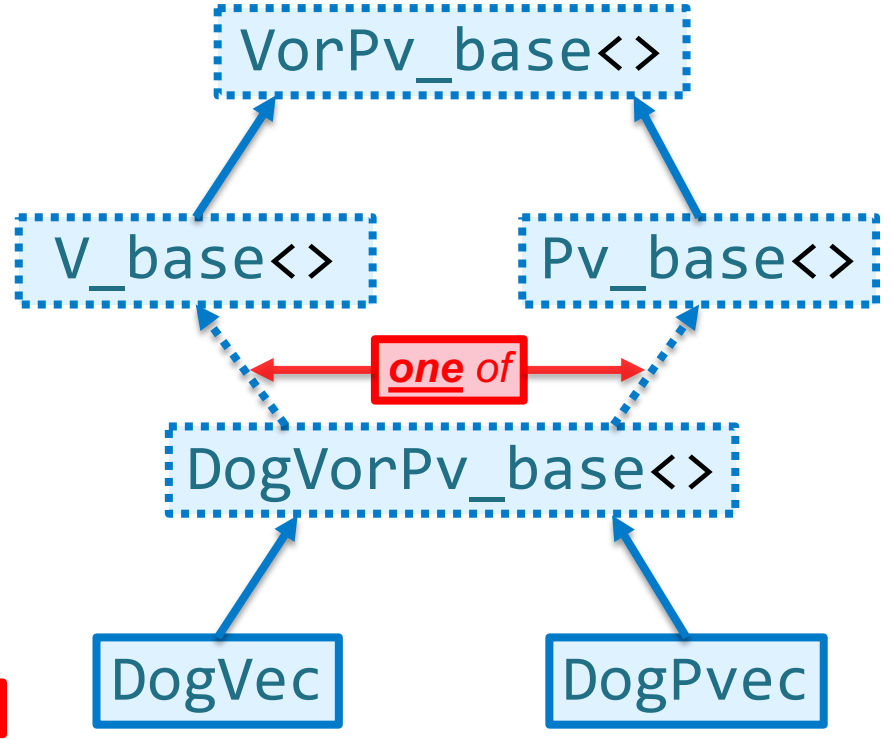
Domain Specific

```
class DogVec : public DogVorPv_base<DogVec, V_base<DogVec,Dog> > {
};
```

Goal: Empty

insert storage spec

```
class DogPvec : public DogVorPv_base<DogPvec, Pv_base<DogPvec, Dog> > {
};
```



`DogVorPv_base<>`: Domain logic for manipulating collections of **`Dog`** instances

`DogVec`: Container of **`Dog`** instances

`DogPvec`: Container of **`Dog*`** instances

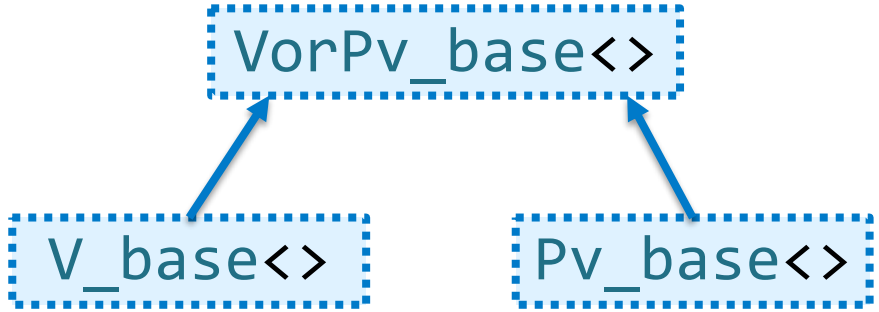
Case Study: Implementation Details

- **C++98** implementation is simple *(CRTP!)*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index){
       // Call implementation in DERIVED
       return static_cast<DERIVED&>(*this).getItem_native_as_ptr_(index);
     }
};
```

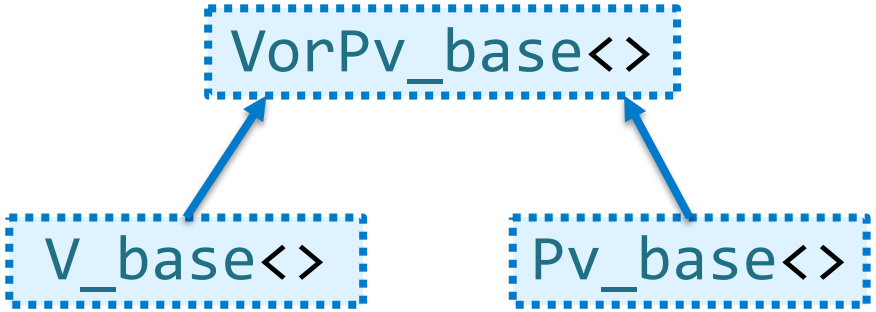
C++98

Implement in DERIVED



Case Study: Implementation Details

- **C++98** implementation is simple *(CRTP!)*



C++98

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index){
       // Call implementation in DERIVED
       return static_cast<DERIVED&>(*this).getItem_native_as_ptr_(index);
     }
};
```

Implement in DERIVED

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
   using BASE = VorPv_base<DERIVED, std::vector<TYPE_ITEM_> >;
   friend class BASE; // Allow BASE access to our protected CRTP-overrides
  protected:
     TYPE_ITEM_* getItem_native_as_ptr_(std::size_t index) {
       return &static_cast<DERIVED&>(*this).getItem_native(index);
     }
};
```

insert storage spec

Need "address-of" element in ***std::vector<Dog>***

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
  using BASE = VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> >;
  friend class BASE; // Allow BASE access to our protected CRTP-overrides
  protected:
     TYPE_ITEM_* getItem_native_as_ptr_(std::size_t index) {
       return static_cast<DERIVED&>(*this).getItem_native(index);
     }
};
```

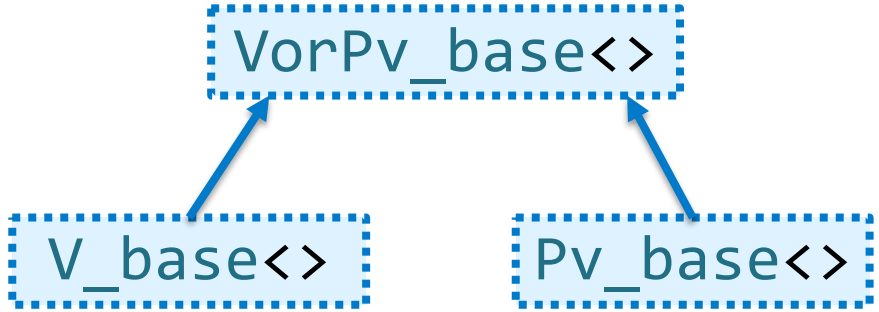
Native element is ptr in ***std::vector<Dog*>***

Case Study: Implementation Details

- **C++17** implementation is easier! (***if constexpr***)

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
    TYPE_ITEM_PTR getItem_as_ptr(std::size_t index){
      // The return statements in a discarded "if constexpr"
      // statement do not participate in function return
      // type deduction (since C++17).
      // The following behavior is well-defined:
     if constexpr(std::is_same<TYPE_ITEM_NATIVE, TYPE_ITEM_PTR>::value)
       return ((DERIVED&)*this).getItem_native(index);
     else
       return &((DERIVED&)*this).getItem_native(index);
    }
};
```

C++17



C++17:
- ***std::is_same<>***
- *Discarded* ***constexpr*** *statements do not participate in type deduction*

Need "address-of" element in ***std::vector<Dog>***

Native element is ptr in ***std::vector<Dog*>***

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

Goal: Empty

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Case Study: In Use

- Container of `Dog`

```
DogVec dog_vec;

dog_vec.addValue_native(Dog("Bella"));
dog_vec.addValue_native(Dog("Oliver"));
dog_vec.addValue_native(Dog("Lucy"));
dog_vec.addValue_native(Dog("Tucker"));

PrintDogs(dog_vec);
```

Output

```
Bella
Oliver
Lucy
Tucker
```

```
void PrintDogs(const DogVec& dog_vec) {

    ??

}
```

Can we implement this?

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
    const std::string& getName(void) const;
};
```

Top 10 Dog Names

of 2018

Male	Female
Max	Bella
Charlie	Lucy
Chopper	Luna
Buddy	Daisy
Jack	Lola
Rocky	Sadie
Duke	Molly
Bear	Bailey
Tucker	Maggie
Oliver	Stella

Case Study: In Use

- Container of `Dog`

```
DogVec dog_vec;

dog_vec.addValue_native(Dog("Bella"));
dog_vec.addValue_native(Dog("Oliver"));
dog_vec.addValue_native(Dog("Lucy"));
dog_vec.addValue_native(Dog("Tucker"));

PrintDogs(dog_vec);
```

Output

```
Bella
Oliver
Lucy
Tucker
```

```
void PrintDogs(const DogVec& dog_vec) {
  for(std::size_t i = 0; i < dog_vec.getNumItems(); ++i) {
    puts(dog_vec.getItem_native(i).getName().c_str());
  }
}
```

"Brute-Force" (raw loop)

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
    const std::string& getName(void) const;
};
```

Top 10 Dog Names

of 2018

Male	Female
Max	Bella
Charlie	Lucy
Chopper	Luna
Buddy	Daisy
Jack	Lola
Rocky	Sadie
Duke	Molly
Bear	Bailey
Tucker	Maggie
Oliver	Stella

Case Study: In Use

- Container of `Dog`

```
DogVec dog_vec;

dog_vec.addValue_native(Dog("Bella"));
dog_vec.addValue_native(Dog("Oliver"));
dog_vec.addValue_native(Dog("Lucy"));
dog_vec.addValue_native(Dog("Tucker"));

PrintDogs(dog_vec);
```

Output

```
Bella
Oliver
Lucy
Tucker
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
    const std::string& getName(void) const;
};
```

Top 10 Dog Names
of 2018

Male	Female
Max	Bella
Charlie	Lucy
Chopper	Luna
Buddy	Daisy
Jack	Lola
Rocky	Sadie
Duke	Molly
Bear	Bailey
Tucker	Maggie
Oliver	Stella

```
void PrintDogs(const DogVec& dog_vec) {
  for(std::size_t i = 0; i < dog_vec.getNumItems(); ++i) {
    puts(dog_vec.getItem_native(i).getName().c_str());
  }
}
```

"Brute-Force" (raw loop)

```
void PrintDogs(const DogVec& dog_vec) {
  const auto& vec_native = dog_vec.get_VorPv_native_ref();
  for(const Dog& item_current : vec_native) {
    puts(item_current.getName().c_str());
  }
}
```

Range-base "`for`" (after accessing internal state)

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

- Is expanded to *(semantically):*

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Loop body
"the good stuff"

- Is expanded to *(semantically):*

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Loop body
"the good stuff"

- Is expanded to *(semantically):*

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Loop body
"the good stuff"

- Is expanded to *(semantically)*:

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

***C++20** (empty in this case)*

***C++11** semantics: Range-based `for` becomes language feature*
***C++17** semantics: `__begin` and `__end` become separate types*
***C++20** semantics: adds init-statement*

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Loop body
"the good stuff"

- Is expanded to *(semantically)*:

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

***C++20** (empty in this case)*

Our container must provide these iterators!

***C++11** semantics: Range-based `for` becomes language feature*
***C++17** semantics: `__begin` and `__end` become separate types*
***C++20** semantics: adds init-statement*

Recall: Range-based `for` loop *(since C++11)*

- Given:

```
const auto& vec_native = ...;
```

- This code:

```
for(const Dog& item_current : vec_native) {
  puts(item_current.getName().c_str());
}
```

Loop body "the good stuff"

- Is expanded to *(semantically)*:

```
init-statement;
auto && __range = vec_native;
auto __begin = begin(__range);
auto __end = end(__range);
for(; __begin != __end; ++__begin) {
  const Dog& item_current = *__begin;
  puts(item_current.getName().c_str());
}
```

***C++20** (empty in this case)*

Our container must provide these iterators!

***C++11** semantics: Range-based `for` becomes language feature*
***C++17** semantics: `__begin` and `__end` become separate types*
***C++20** semantics: adds init-statement*

To Make Your Container work with Range-based `for`

*(pick **one** of):*

1. **Create member `x:begin()` and `x::end()`** *(returning something that acts like an iterator)*
2. **Create a free function `begin(x&)` and `end(x&)`** in the same namespace as your type `'x'` *(returning something that acts like an iterator)*

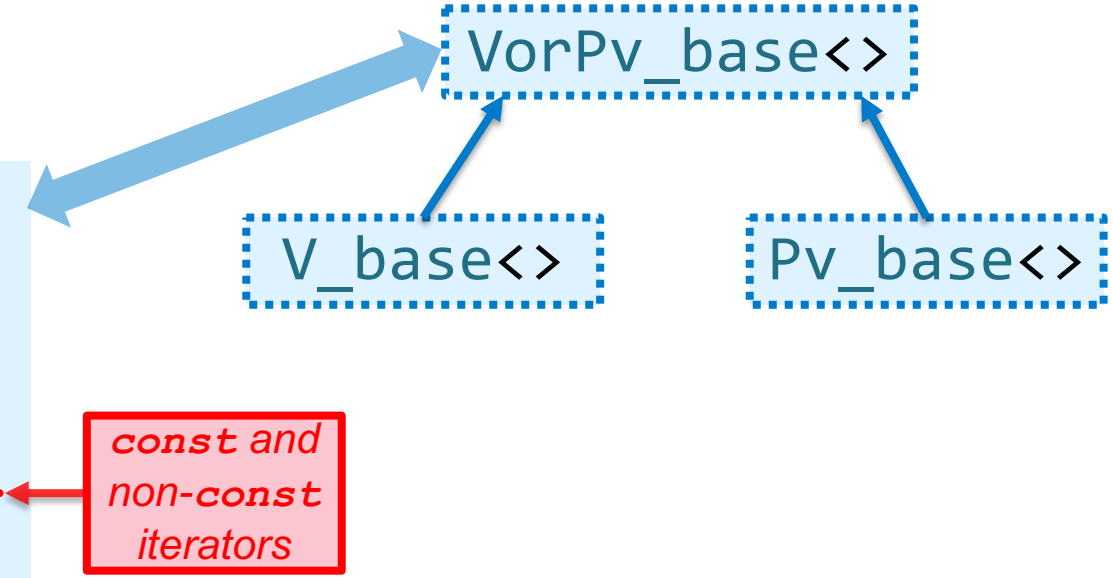
Also, do it for `const` versions

https://stackoverflow.com/questions/8164567/how-to-make-my-custom-type-to-work-with-range-based-for-loops/31457319

Supporting Range-based `for` loop *(since C++11)*

- Our "container" happens to wrap a `std::` container *(that does all the iterator work):*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  protected:
    VorPv_native_ vorpv_native_;
  public:
   auto begin(void) const { return vorpv_native_.begin(); }
   auto begin(void)       { return vorpv_native_.begin(); }
   auto end(void) const   { return vorpv_native_.end(); }
   auto end(void)         { return vorpv_native_.end(); }
};
```



previously:

```
void PrintDogs(const DogVec& dog_vec) {
  const auto& vec_native = dog_vec.get_VorPv_native_ref();
  for(const Dog& item_current : vec_native) {
    puts(item_current.getName().c_str());
  }
}
```

now:

```
void PrintDogs(const DogVec& dog_vec) {
  for(const Dog& item_current : dog_vec) {
    puts(item_current.getName().c_str());
  }
}
```

The **C++ Language Features** and **`std::` algorithms** **are designed** *(and intended!)* **to integrate** *(transparently!)* **with your** *(custom! Domain-specific!)* **containers**

They are BEGGING you!
Please Do This!

Making More Efficient: `emplace_back()`

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
  protected:
    VorPv_native vorpv_native_;
  public:
    void addValue_native(TYPE_ITEM_NATIVE item_value_native){
      vorpv_native_.push_back(item_value_native);
    }

    template<typename... ARGS_CTOR>
    void addValue_native_emplace(ARGS_CTOR&&... args_ctor) {
      vorpv_native_.emplace_back(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
};
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
    const std::string& getName(void) const;
};
```

Making More Efficient: `emplace_back()`

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
  public:
    using TYPE_ITEM_NATIVE      = typename VorPv_native_::value_type;
  protected:
    VorPv_native vorpv_native_;
  public:
      void addValue_native(TYPE_ITEM_NATIVE item_value_native){
        vorpv_native_.push_back(item_value_native);
      }

      template<typename... ARGS_CTOR>
      void addValue_native_emplace(ARGS_CTOR&&... args_ctor) {
        vorpv_native_.emplace_back(
          std::forward<ARGS_CTOR&&>(args_ctor)...);
      }
};
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
    const std::string& getName(void) const;
};
```

previously:

```
DogVec dog_vec;

dog_vec.addValue_native(Dog("Bella"));
dog_vec.addValue_native(Dog("Oliver"));
dog_vec.addValue_native(Dog("Lucy"));
dog_vec.addValue_native(Dog("Tucker"));

PrintDogs(dog_vec);
```

Copy ctor invoked

now:

```
DogVec dog_vec;

dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

PrintDogs(dog_vec);
```

Args forwarded to `Dog` ctor

Output

```
Bella
Oliver
Lucy
Tucker
```

Stupid Pet Tricks: Container Manipulation

- What can we ***do*** with this container abstraction?

Skateboarding Dog - HD Redux

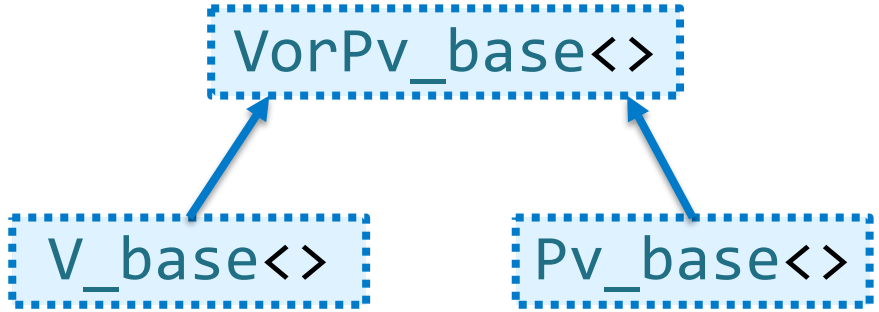
5,244,610 views

https://www.youtube.com/watch?v=R8XAlSp838Y

Get As... *pvec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:




   template<class TYPE_PVEC>
   void getAs_pvec(TYPE_PVEC& as_pvec) const {

         ??

   }

   template<class TYPE_PVEC>
   TYPE_PVEC getAs_pvec(void) const {

         ??

   }
};
```

Can we implement these?



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

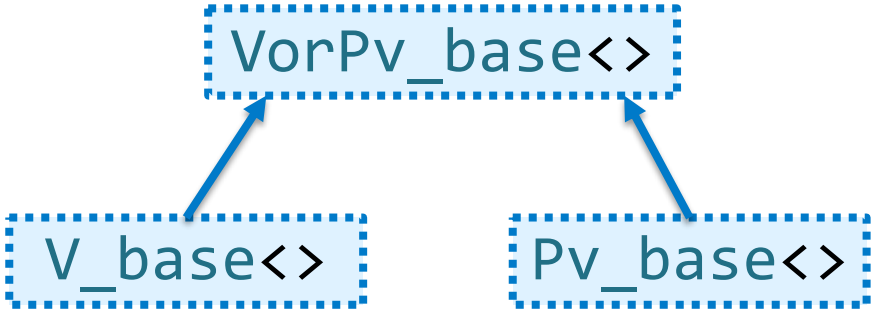
```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... *pvec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:




   template<class TYPE_PVEC>
   void getAs_pvec(TYPE_PVEC& as_pvec) const {
     as_pvec.clear();
     static_cast<const DERIVED&>(*this).appendTo_pvec(as_pvec);
   }

   template<class TYPE_PVEC>
   TYPE_PVEC getAs_pvec(void) const {

         ??

   }
};
```

Pretend this exists



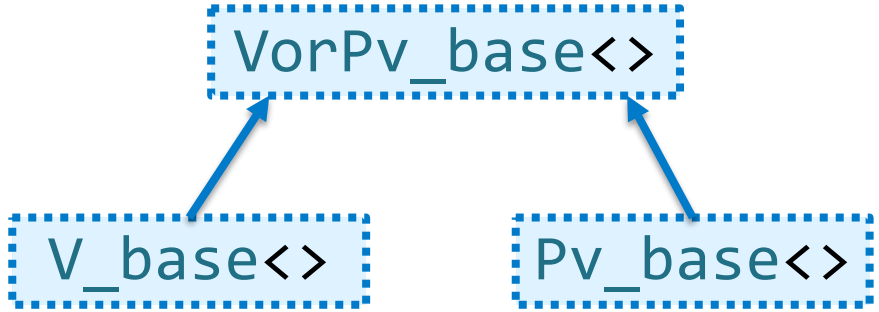
Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... *pvec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& as_pvec) const {
          ??   Can we implement this?
    }

    template<class TYPE_PVEC>
    void getAs_pvec(TYPE_PVEC& as_pvec) const {
      as_pvec.clear();
      static_cast<const DERIVED&>(*this).appendTo_pvec(as_pvec);
    }

    template<class TYPE_PVEC>
    TYPE_PVEC getAs_pvec(void) const {

          ??

    }
};
```



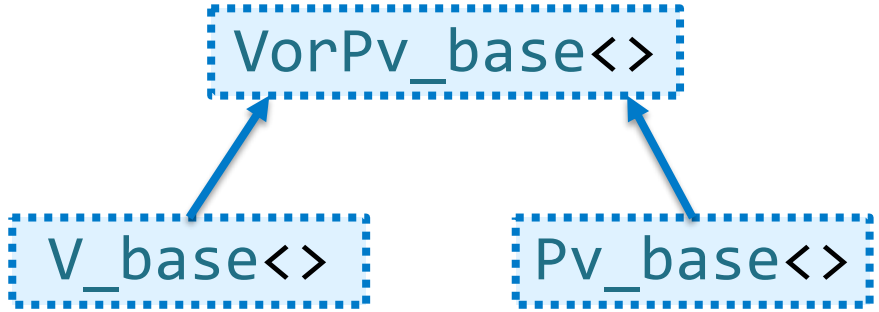
Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... *pvec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& as_pvec) const {
          ??  Can we implement this?
    }

    template<class TYPE_PVEC>
    void getAs_pvec(TYPE_PVEC& as_pvec) const {
      as_pvec.clear();
      static_cast<const DERIVED&>(*this).appendTo_pvec(as_pvec);
    }

    template<class TYPE_PVEC>
    TYPE_PVEC getAs_pvec(void) const {
      TYPE_PVEC as_pvec;
      static_cast<const DERIVED&>(*this).appendTo_pvec(as_pvec);
      return as_pvec;
    }
};
```



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

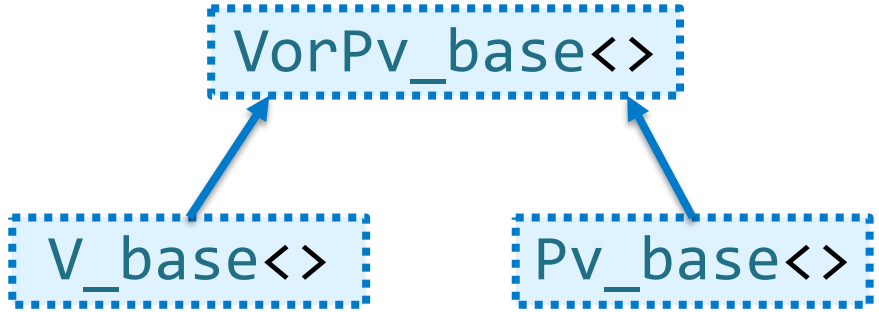
```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... *pvec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& as_pvec) const {
      std::size_t num_items =
        static_cast<const DERIVED&>(*this).getNumItems();

      for(std::size_t i = 0; i < num_items; ++i) {
        pvec_append_to.addValue_native(
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i));
      }
    }

    template<class TYPE_PVEC>
    void getAs_pvec(TYPE_PVEC& as_pvec) const;

    template<class TYPE_PVEC>
    TYPE_PVEC getAs_pvec(void) const;

};
```

We "know" a pvec natively stores "ptrs"

We "know" how to get our element "as-ptr"



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

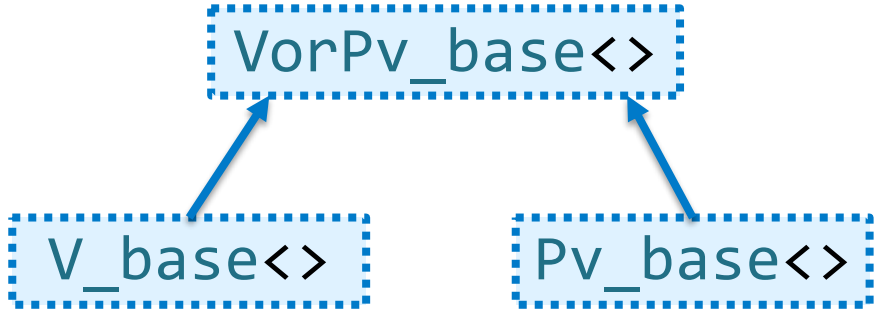
```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... *pvec* + "Short-Circuit"

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& as_pvec) const {
      std::size_t num_items =
        static_cast<const DERIVED&>(*this).getNumItems();

      for(std::size_t i = 0; i < num_items; ++i) {
        pvec_append_to.addValue_native(
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i));
      }
    }

    template<class TYPE_PVEC>
    void getAs_pvec(TYPE_PVEC& as_pvec) const;

    template<class TYPE_PVEC>
    TYPE_PVEC getAs_pvec(void) const;

};
```

We "know" a pvec natively stores "ptrs"

We "know" how to get our element "as-ptr"



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**pvec**"…

Note: If our container ***is*** a ***pvec***, merely return ****this***

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

"Do Nothing" optimization

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
  using BASE = VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> >;
  public:
    using BASE::getAs_pvec;
    const DERIVED& getAs_pvec(void) const {
      return static_cast<const DERIVED&>(*this);
    }
};
```

Playing With Our Pets

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to ‘dog_pvec’)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)
```

*"Do Nothing" optimization (returns **`*this`**)*

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);
```

*"Do Nothing" optimization (returns **`*this`**)*

previously:

```
void PrintDogs(const DogVec& dog_vec) {
  for(const Dog& item_current : dog_vec) {
    puts(item_current.getName().c_str());
  }
}
```

now (add overload):

```
void PrintDogs(const DogPvec& dog_pvec) {
  for(const Dog* item_current : dog_pvec) {
    if(item_current)
      puts(item_current->getName().c_str());
    else
      puts("<null>");
  }
}
```

Output

```
Bella
Oliver
Lucy
Tucker
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);
```

*"Do Nothing" optimization (returns **`*this`**)*

How about providing a **single implementation** *(for **`vec`** or **`pvec`**)*?

previously:

```
void PrintDogs(const DogVec& dog_vec) {
  for(const Dog& item_current : dog_vec) {
    puts(item_current.getName().c_str());
  }
}
```

```
void PrintDogs(const DogPvec& dog_pvec) {
  for(const Dog* item_current : dog_pvec) {
    if(item_current)
      puts(item_current->getName().c_str());
    else
      puts("<null>");
  }
}
```

Single implementation:

`vec` *or* **`pvec`**

```
template<class DOG_VorPv>
void PrintDogs(const DOG_VorPv& dog_vorpv) {
  const Dog* item_current;
  std::size_t num_items = dog_vorpv.getNumItems();
  for(std::size_t i = 0; i < num_items; ++i) {
    if((item_current = dog_vorpv.getItem_as_ptr(i)))
      puts(item_current->getName().c_str());
    else
      puts("<null>");
  }
}
```

More Stupid Pet Tricks?

- Review:
  - We have **different types** for `DogVec` and `DogPvec`
  - Anything can present "**as-if" it were a `pvec`**

More Stupid Pet Tricks?

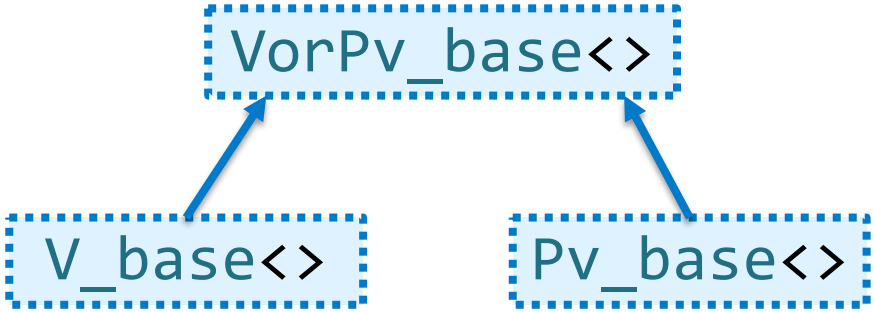
- Review:
  - We have **different types** for `DogVec` and `DogPvec`
  - Anything can present "**as-if" it were a `pvec`**

- Can we ***"re-constitute"*** a `vec` from a `pvec`?
  - *Note:* This **implies a "deep-copy"** or "clone" *(because* **`vec`** ***wholly contains*** *its instances)*

Get As... *vec*

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_VEC>
    void appendTo_vec(TYPE_VEC& vec_append_to) const {
      std::size_t num_items =
       static_cast<const DERIVED&>(*this).getNumItems();
      TYPE_ITEM_CONST_PTR item_current;
      for(std::size_t i = 0; i < num_items; ++i) {
        item_current =
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i);
        if(item_current)
          vec_append_to.addValue_native_emplace(*item_current);
        else // 'nullptr' entry, add empty object
          vec_append_to.addValue_native_emplace();
      }
    }
    template<class TYPE_VEC>
      void getAs_vec(TYPE_VEC& as_vec) const;
    template<class TYPE_VEC>
      TYPE_VEC getAs_vec(void) const;
};
```

*Or, could implicitly "skip" **nullptr** entries*



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**vec**"…

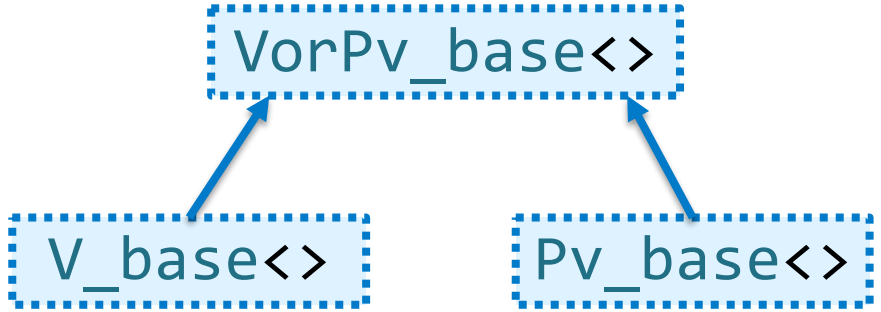
```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Get As... `vec` + "Short-Circuit"

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    template<class TYPE_VEC>
    void appendTo_vec(TYPE_VEC& vec_append_to) const {
      std::size_t num_items =
       static_cast<const DERIVED&>(*this).getNumItems();
      TYPE_ITEM_CONST_PTR item_current;
      for(std::size_t i = 0; i < num_items; ++i) {
        item_current =
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i);
        if(item_current)
          vec_append_to.addValue_native_emplace(*item_current);
        else // 'nullptr' entry, add empty object
          vec_append_to.addValue_native_emplace();
      }
    }
    template<class TYPE_VEC>
      void getAs_vec(TYPE_VEC& as_vec) const;
    template<class TYPE_VEC>
      TYPE_VEC getAs_vec(void) const;
};
```

*Or, could implicitly "skip" **`nullptr`** entries*



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "**`vec`**"…

Note: If our container ***is*** a **`vec`**, merely return **`*this`**

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

"Do Nothing" optimization

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
  using BASE = VorPv_base<DERIVED, std::vector<TYPE_ITEM_> >;
  public:
    using BASE::getAs_vec;
    const DERIVED& getAs_vec(void) const {
      return static_cast<const DERIVED&>(*this);
    }
};
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to ‘dog_pvec’)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Output

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec   = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Output *(x2)*

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);

DogVec     dog_vec_copy = dog_pvec.getAs_vec();    // ok (get deep-copy)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Output *(x2)*

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);

DogVec     dog_vec_copy = dog_pvec.getAs_vec();    // ok (get deep-copy)

const DogVec& dog_vec_0 = dog_pvec.getAs_vec();    // ok (reference lifetime extension)
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Output *(x2)*

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);

DogVec     dog_vec_copy = dog_pvec.getAs_vec();    // ok (get deep-copy)

const DogVec& dog_vec_0 = dog_pvec.getAs_vec();    // ok (reference lifetime extension)
const DogVec& dog_vec_1 = dog_vec.getAs_vec();     // ok (reference to 'dog_vec')
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

Output *(x2)*

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);

DogVec     dog_vec_copy = dog_pvec.getAs_vec();    // ok (get deep-copy)

const DogVec& dog_vec_0 = dog_pvec.getAs_vec();    // ok (reference lifetime extension)
const DogVec& dog_vec_1 = dog_vec.getAs_vec();     // ok (reference to 'dog_vec')
assert(&dog_vec_1 == &dog_vec);                    // ok (same instance)
```

"Do Nothing" optimization (returns ***this**)

"Do Nothing" optimization (returns ***this**)

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

Output *(x2)*

```
Bella
Oliver
Lucy
Tucker
```

Playing With Our Pets *(again!)*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella");
dog_vec.addValue_native_emplace("Oliver");
dog_vec.addValue_native_emplace("Lucy");
dog_vec.addValue_native_emplace("Tucker");

DogPvec        dog_pvec    = dog_vec.getAs_pvec();  // ok (get copy)

const DogPvec& dog_pvec_0 = dog_vec.getAs_pvec();  // ok (reference lifetime extension)
const DogPvec& dog_pvec_1 = dog_pvec.getAs_pvec(); // ok (reference to 'dog_pvec')
assert(&dog_pvec_1 == &dog_pvec);                  // ok (same instance)

PrintDogs(dog_vec);
PrintDogs(dog_pvec);

DogVec     dog_vec_copy = dog_pvec.getAs_vec();    // ok (get deep-copy)

const DogVec& dog_vec_0 = dog_pvec.getAs_vec();    // ok (reference lifetime extension)
const DogVec& dog_vec_1 = dog_vec.getAs_vec();     // ok (reference to 'dog_vec')
assert(&dog_vec_1 == &dog_vec);                    // ok (same instance)

PrintDogs(dog_vec_copy);
PrintDogs(dog_vec_0);
PrintDogs(dog_vec_1);
```

```
class Dog {
  private:
    std::string str_name_;
  public:
    explicit Dog(const std::string& str_name);
};
```

*"Do Nothing" optimization (returns **`*this`**)*

*"Do Nothing" optimization (returns **`*this`**)*

Transparent manipulation between **`vec`** and **`pvec`**

Output *(x5)*

```
Bella
Oliver
Lucy
Tucker
```

Review: Stupid Pet Tricks

- We have **different types** for `DogVec` and `DogPvec`
  - `vec:` *wholly contains* `Dog` instances
  - `pvec:` *references* `Dog` instances

Review: Stupid Pet Tricks

- We have **different types** for `DogVec` and `DogPvec`
  - `vec:` *wholly contains* `Dog` instances
  - `pvec:` *references* `Dog` instances
- Anything can present "**as-if" it were a `pvec`**
  - `vec:` creates *container-of-pointers* referencing `vec` members
  - `pvec:` returns `*this`

Review: Stupid Pet Tricks

- We have **different types** for **`DogVec`** and **`DogPvec`**
  - **`vec:`** *wholly contains* **`Dog`** instances
  - **`pvec:`** *references* **`Dog`** instances
- Anything can present "**as-if" it were a `pvec`**
  - **`vec:`** creates *container-of-pointers* referencing **`vec`** members
  - **`pvec:`** returns **`*this`**
- Anything can present "**as-if" it were a `vec`**
  - **`vec:`** returns **`*this`**
  - **`pvec:`** creates *container of "deep copy" (clone)* objects from **`pvec`** references

Slices, Subsetting, and Data Reduction

Getting What You Want

Getting What You Want

Common Container Goal:
Extract the meta-data desired

- How is this done?

Algorithm A:

1. **Find The One** matching item
   - Perhaps is not present
2. **Reduce** to the meta-data desired, *(examples):*
   - Extract field(s)
   - Forward to other operation
   - Perform side-effect *(print, mutate, etc.)*

Algorithm B:

1. **Find The Several** matching items
   - Perhaps none are present
2. **Reduce** to the meta-data desired, *(examples):*
   - Extract field(s)
   - Forward to other operation
   - Perform side-effect *(print, mutate, etc.)*

Do you notice any similarities?

Data Reduction

Data Reduction *(def):*
The transformation of data into a simplified form through corrections and ordering

Slice *(def):*
- *(verb):* To extract a subset collection
- *(noun):* A subset collection

- ***Why?***
  - Data Reduction **extracts meaningful parts** from *(large and noisy)* data sets

Data Reduction

Data Reduction *(def):*
The transformation of data into a simplified form through corrections and ordering

Slice *(def):*
- *(verb):* To extract a subset collection
- *(noun):* A subset collection

- ***Why?***
  - Data Reduction **extracts meaningful parts** from *(large and noisy)* data sets

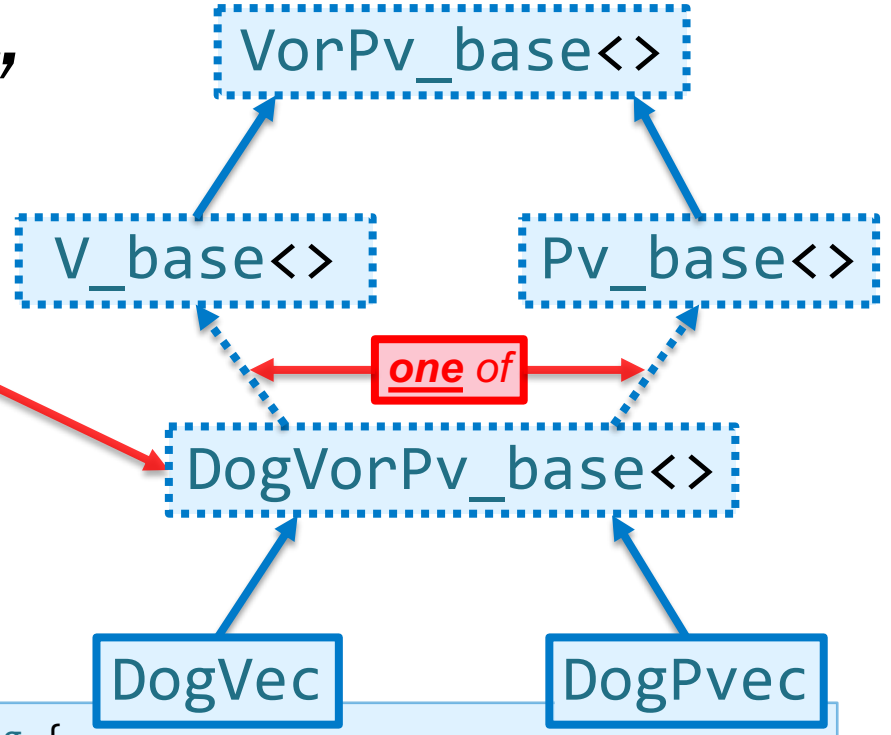
- ***How?***
  1. **Data slicing** *(i.e., "ordering")*
  2. **Data cleaning** *(i.e., "filtering" and "repair-or-replacement")*

Subsetting Our Dogs: *"Oldest One"*

```
template<class DERIVED, class TYPE_VorPv_base>
class DogVorPv_base : public TYPE_VorPv_base {
  public:
    Dog* get_oldest_pickOne(void) {
      if(!static_cast<DERIVED&>(*this).hasState())
        return nullptr;
      Dog* dog_oldest = static_cast<DERIVED&>(*this).getItem_as_ptr(0);
      std::size_t num_items = static_cast<DERIVED&>(*this).getNumItems();
      Dog* dog_current;
      for(std::size_t i = 1; i < num_items; ++i) {
        dog_current = static_cast<DERIVED&>(*this).getItem_as_ptr(i);
        if(dog_current) {
          if(!dog_oldest) {
            dog_oldest = dog_current;
          } else {
            if(dog_current->getAge() > dog_oldest->getAge()) {
              dog_oldest = dog_current;
            }
          }
        }
      }
      return dog_oldest;
    }
};
```

Goal: Domain logic

Trivia: Algorithm picks "first-oldest"

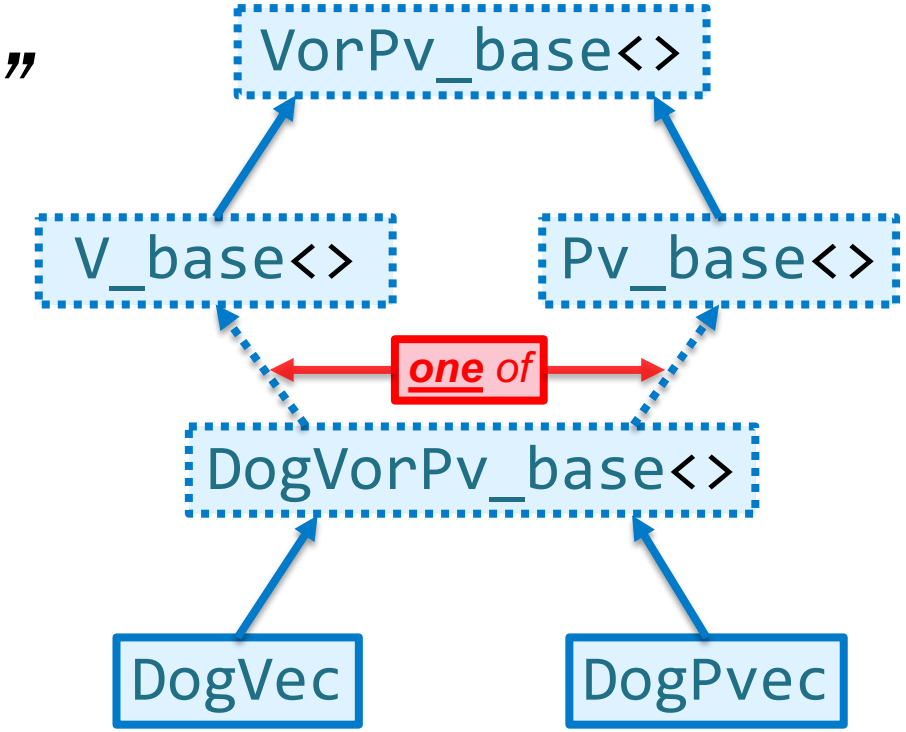



```
class Dog {
  private:
    std::string str_name_;
    DogAge      dog_age_;
  public:
    Dog(void) = default;
    explicit Dog(
      const std::string& str_name,
      DogAge dog_age);
    Dog(const Dog& other) = default;
    Dog& operator=(const Dog& other) = default;
    DogAge getAge(void) const;
    const std::string& getName(void) const;
};
```

Add DogAge

Subsetting Our Dogs: *"Oldest One"*

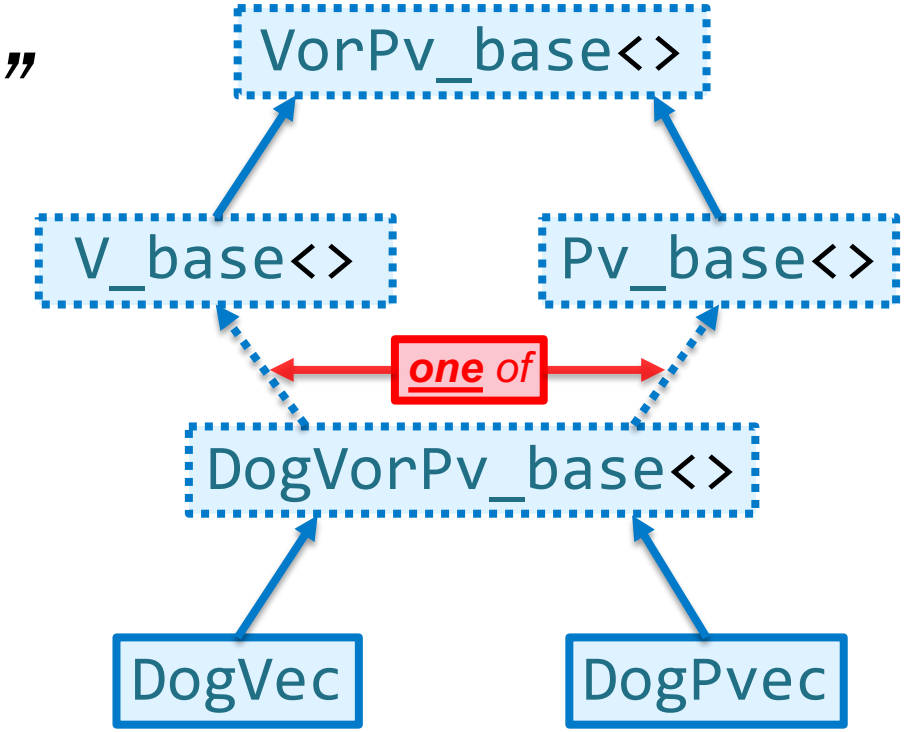
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();
```



Subsetting Our Dogs: *"Oldest One"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

Dog* dog_oldest0 = dog_vec.get_oldest_pickOne();
```

Ask ***vec*** *for oldest*

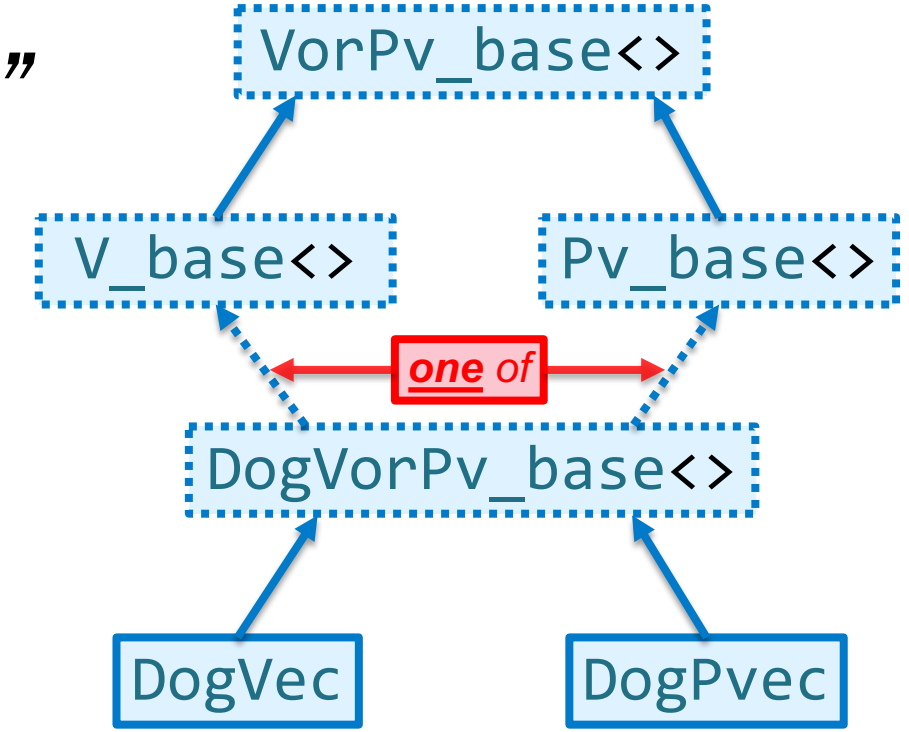


Subsetting Our Dogs: *"Oldest One"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

Dog* dog_oldest0 = dog_vec.get_oldest_pickOne();
Dog* dog_oldest1 = dog_pvec.get_oldest_pickOne();
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*



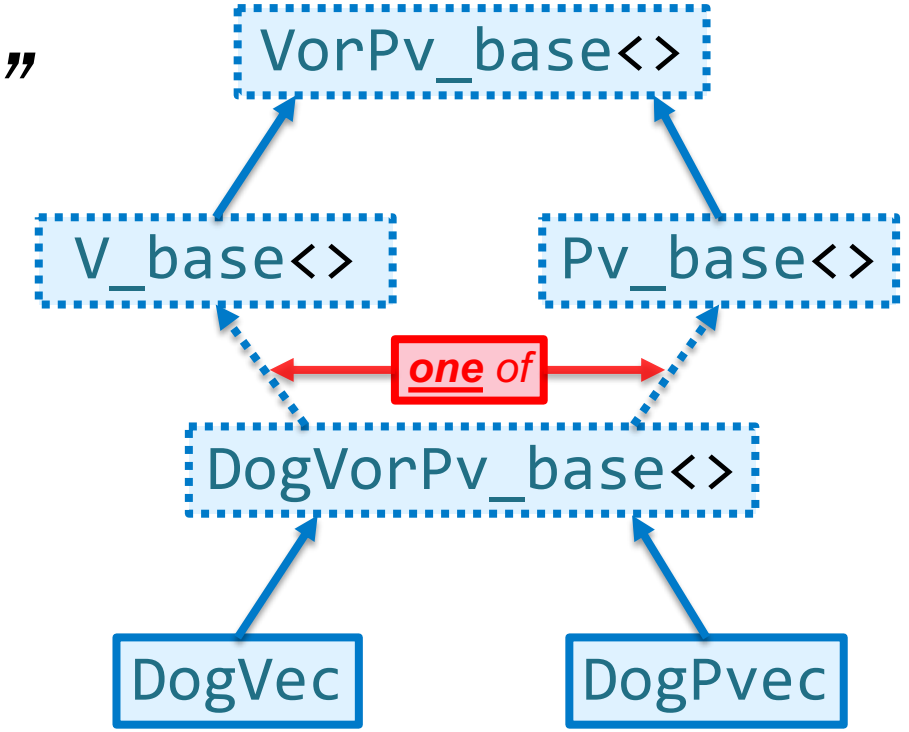
Subsetting Our Dogs: *"Oldest One"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

Dog* dog_oldest0 = dog_vec.get_oldest_pickOne();
Dog* dog_oldest1 = dog_pvec.get_oldest_pickOne();

assert(dog_oldest0);
assert(dog_oldest0 == dog_oldest1);
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

They agree on same "oldest" instance! (they share the same implementation)



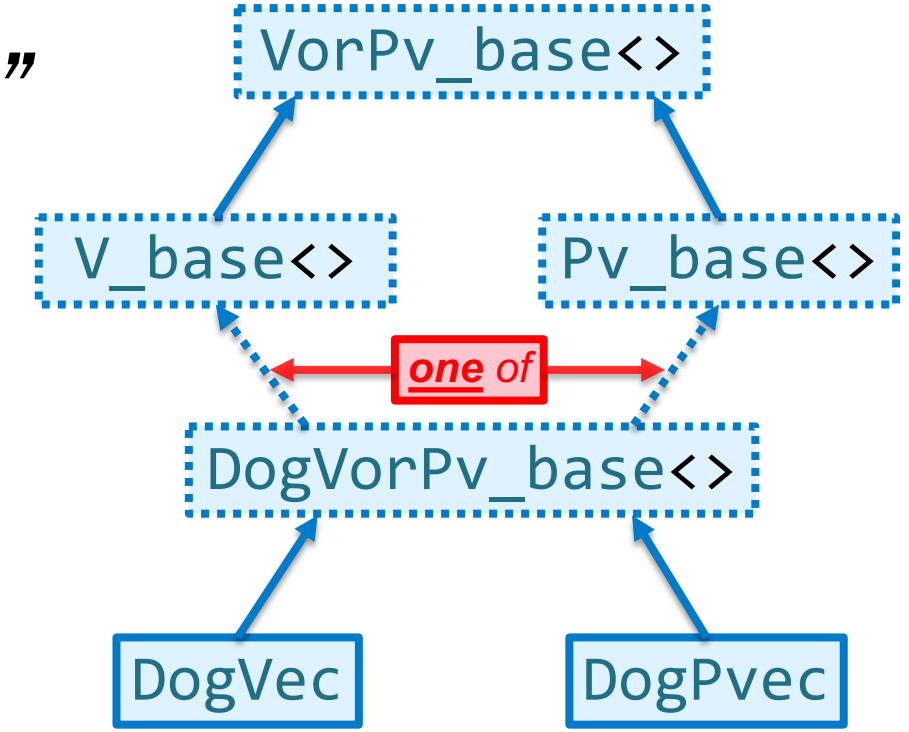
Subsetting Our Dogs: *"Oldest One"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

Dog* dog_oldest0 = dog_vec.get_oldest_pickOne();
Dog* dog_oldest1 = dog_pvec.get_oldest_pickOne();

assert(dog_oldest0);
assert(dog_oldest0 == dog_oldest1);

PrintDogs(dog_vec);
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

They agree on same "oldest" instance! (they share the same implementation)



Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```

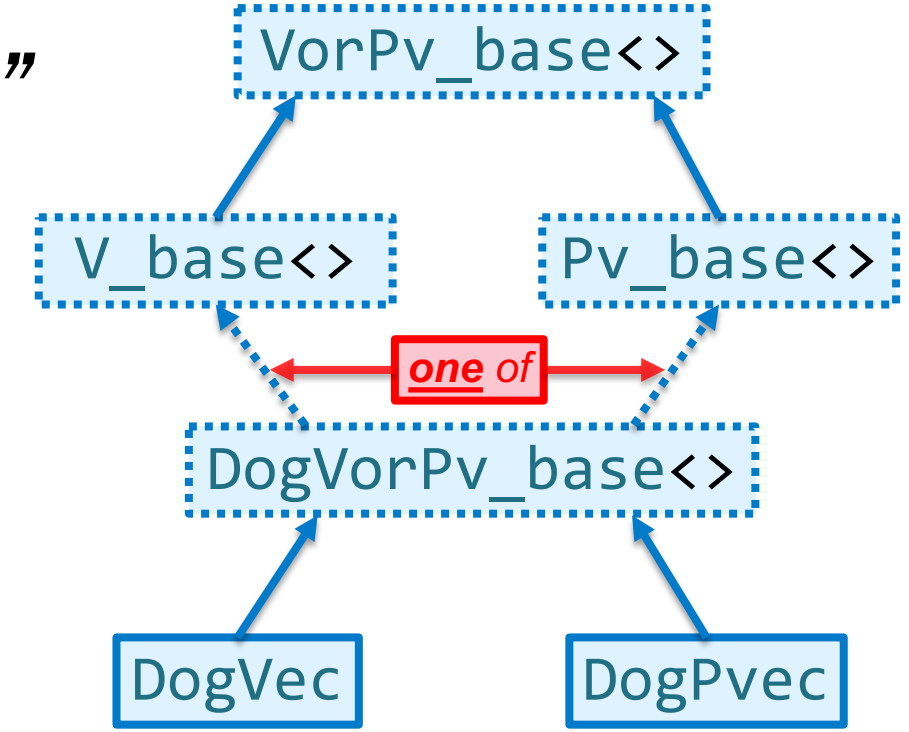
Subsetting Our Dogs: *"Oldest One"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

Dog* dog_oldest0 = dog_vec.get_oldest_pickOne();
Dog* dog_oldest1 = dog_pvec.get_oldest_pickOne();

assert(dog_oldest0);
assert(dog_oldest0 == dog_oldest1);

PrintDogs(dog_vec);
std::string str_msg("...Oldest (one): ");
str_msg += dog_oldest0->getName();
puts(str_msg.c_str());
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

They agree on same "oldest" instance! (they share the same implementation)



Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
...Oldest (one): Bella
```

Slicing On A Predicate

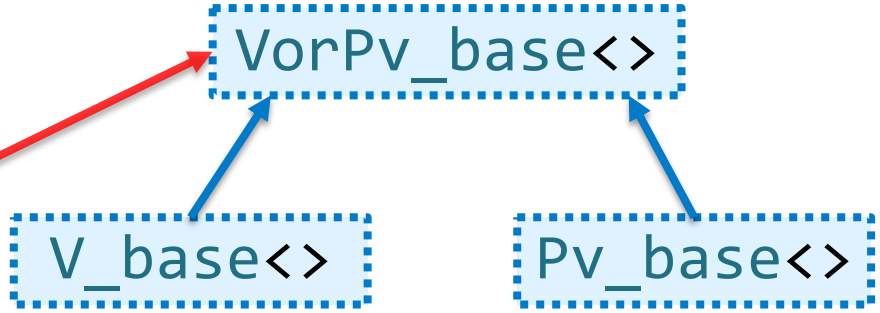
- We are able to **implement a single algorithm** that works on both `vec` and `pvec`

Slicing On A Predicate

- We are able to **implement a single algorithm** that works on both `vec` and `pvec`

- Can we **inject localized context into a generic "slice"** *(i.e., "sub-setting")* function?

Slicing On A Predicate



Goal: Generic logic

Test Dog*

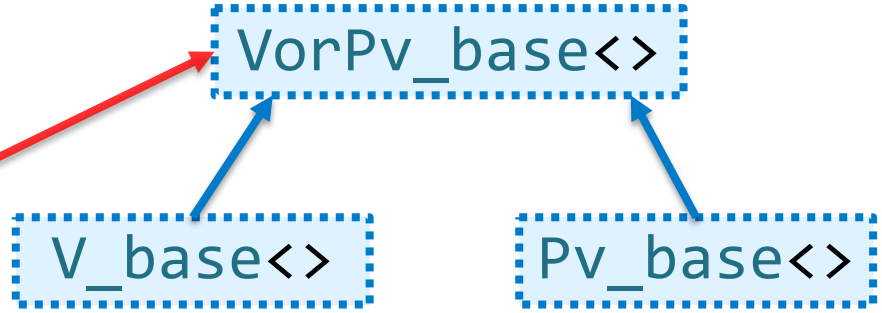
Test Dog&

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    //    |-------
    //    |using CALLABLE_IS_MATCH_PTR = (const ITEM*)->bool
    //    |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_PTR>
    void appendToPvec_matches_each_ptr(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_PTR callable_is_match_each_ptr) const;

    //    |-------
    //    |using CALLABLE_IS_MATCH_REF = (const ITEM&)->bool
    //    |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_REF>
    void appendToPvec_matches_each_ref(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_REF callable_is_match_each_ref) const;
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Slicing On A Predicate



Goal: Generic logic

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    //    |-------
    //    |using CALLABLE_IS_MATCH_PTR =(const ITEM*)->bool
    //    |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_PTR>
    void appendToPvec_matches_each_ptr(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_PTR callable_is_match_each_ptr) const;

    //    |-------
    //    |using CALLABLE_IS_MATCH_REF =(const ITEM&)->bool
    //    |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_REF>
    void appendToPvec_matches_each_ref(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_REF callable_is_match_each_ref) const;
};
```

Test Dog*

Test Dog&

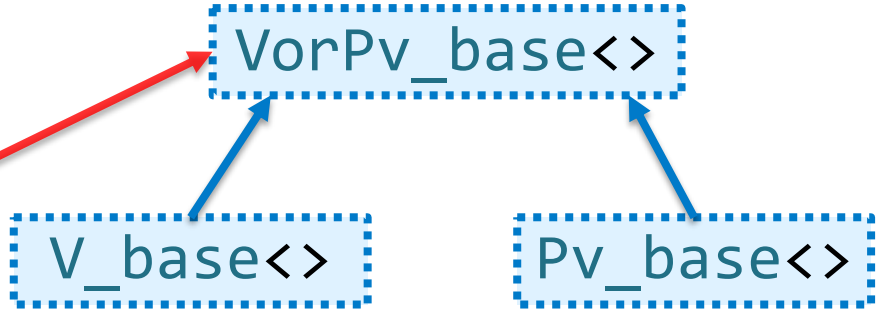
```
template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_PTR>
void getPvec_matches_each_ptr(
  ITEM_PVEC& item_pvec_matches,
  CALLABLE_IS_MATCH_PTR callable_is_match_each_ptr) const
{
  item_pvec_matches.clear();
  static_cast<const DERIVED&>(*this).appendToPvec_matches_each_ptr(
    item_pvec_matches, callable_is_match_each_ptr);
}

template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_REF>
void getPvec_matches_each_ref(
  ITEM_PVEC& item_pvec_matches,
  CALLABLE_IS_MATCH_REF callable_is_match_each_ref) const
{
  item_pvec_matches.clear();
  static_cast<const DERIVED&>(*this).appendToPvec_matches_each_ref(
    item_pvec_matches, callable_is_match_each_ref);
}
```



```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Slicing On A Predicate



Goal: Generic logic

Test Dog*

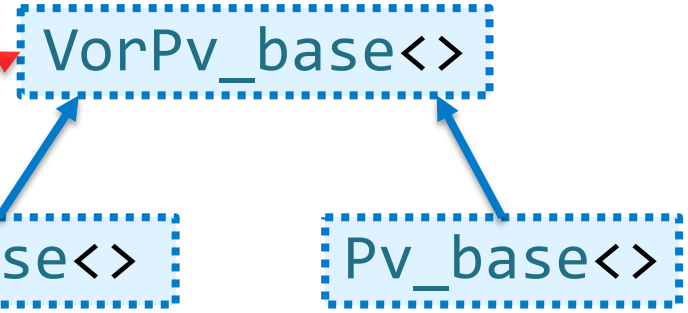
```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    //   |-------
    //   |using CALLABLE_IS_MATCH_PTR = (const ITEM*)->bool
    //   |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_PTR>
    void appendToPvec_matches_each_ptr(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_PTR callable_is_match_each_ptr) const
    {
      std::size_t num_items =
      static_cast<const DERIVED&>(*this).numItems();
      for(long i = 0; i < num_items; ++i)   {
        if(callable_is_match_each_ptr(
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i))) {
          item_pvec_match.addValue_native(item_current);
        }
      }
    }

};
```

Similar implementation to previous examples

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Slicing On A Predicate



Goal: Generic logic

```
template<class DERIVED, class VorPv_native_>
class VorPv_base {
    ...
  public:
    //   |-------
    //   |using CALLABLE_IS_MATCH_PTR = (const ITEM*)->bool
    //   |-------
    template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_PTR>
    void appendToPvec_matches_each_ptr(
      ITEM_PVEC& item_pvec_matches,
      CALLABLE_IS_MATCH_PTR callable_is_match_each_ptr) const
    {
      std::size_t num_items =
      static_cast<const DERIVED&>(*this).numItems();
      for(long i = 0; i < num_items; ++i)   {
        if(callable_is_match_each_ptr(
          static_cast<const DERIVED&>(*this).getItem_as_ptr(i))) {
          item_pvec_match.addValue_native(item_current);
        }
      }
    }
};
```

Test Dog*

Similar implementation to previous examples

```
template<class DERIVED, class TYPE_ITEM_>
class V_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_> > {
};
```

```
//   |-------
//   |using CALLABLE_IS_MATCH_REF = (const ITEM&)->bool
//   |-------
template<class ITEM_PVEC, typename CALLABLE_IS_MATCH_REF>
void appendToPvec_matches_each_ref(
  ITEM_PVEC& item_pvec_matches,
  CALLABLE_IS_MATCH_REF callable_is_match_each_ref) const
{
  std::size_t num_items =
  static_cast<const DERIVED&>(*this).getNumItems();
  TYPE_ITEM_CONST_PTR item_current;
  for(long i = 0; i < num_items; ++i)   {
    if((item_current =
      static_cast<const DERIVED&>(*this).getItem_as_ptr(i))) {
      if(callable_is_match_each_ref(*item_current)) {
        item_pvec_match.addValue_native(item_current);
      }
    }
  }
}
```

Test Dog&

If assignment to **item_current** is non-**nullptr**, then do indirection

```
template<class DERIVED, class TYPE_ITEM_>
class Pv_base : public VorPv_base<DERIVED, std::vector<TYPE_ITEM_*> > {
};
```

Subsetting Our Dogs: *"Oldest Several"*

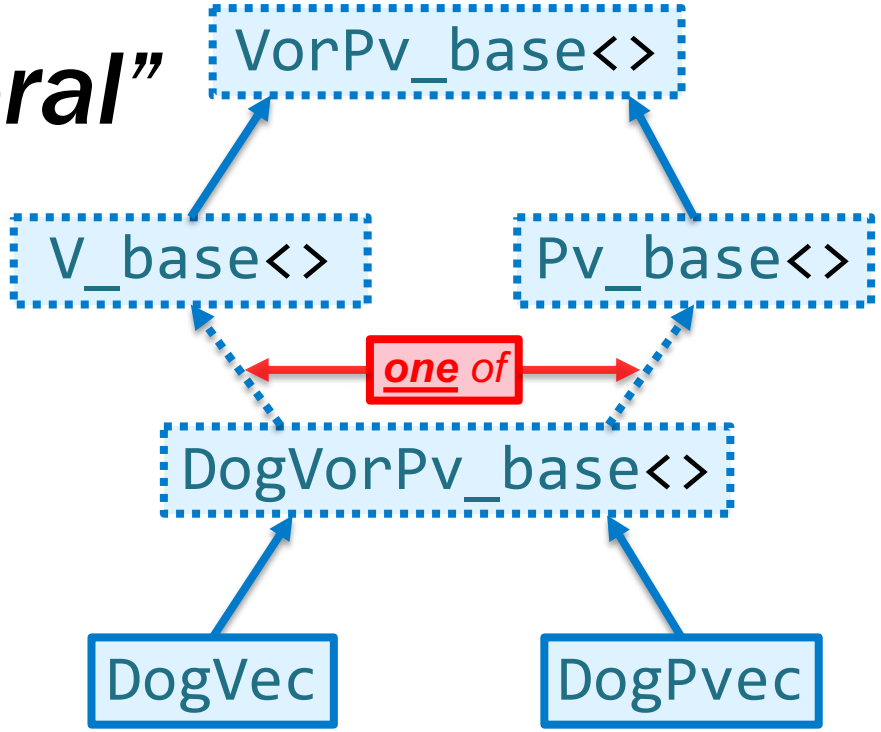
*Get the (several) "oldest" instances (where several dogs share the same "oldest" **DogAge**)*

```
class DogPvec;

template<class DERIVED, class TYPE_VorPv_base>
class DogVorPv_base : public TYPE_VorPv_base {
  public:
    DogPvec get_oldest(void) const;

      void get_oldest(DogPvec& dog_pvec) const {
        dog_pvec.clear();
        if(!static_cast<const DERIVED&>(*this).hasState())
          return;
        const Dog* dog_oldest =
          static_cast<const DERIVED&>(*this).get_oldest_pickOne();
        if(dog_oldest)   {
          DogAge age_search_for = dog_oldest->getAge();
          static_cast<const DERIVED&>(*this).appendToPvec_matches_each_ref(
            dog_pvec,
            [&](const Dog& item_current){
              return (item_current.getAge() == age_search_for); }
          );
        }
      }
};
```

implementation calls

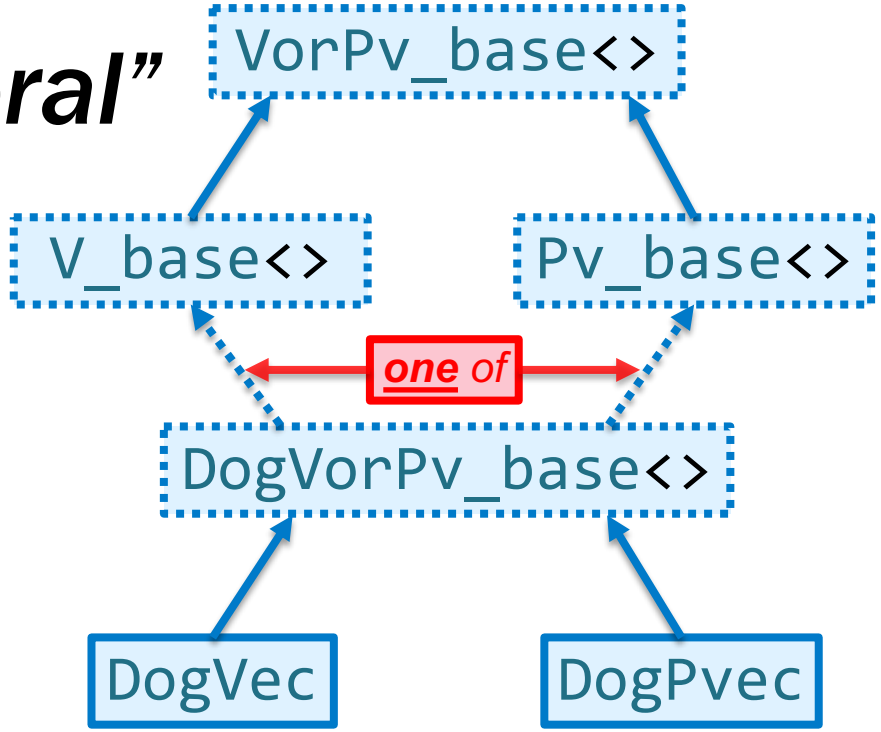
predicate

Slice on a predicate



Subsetting Our Dogs: *"Oldest Several"*

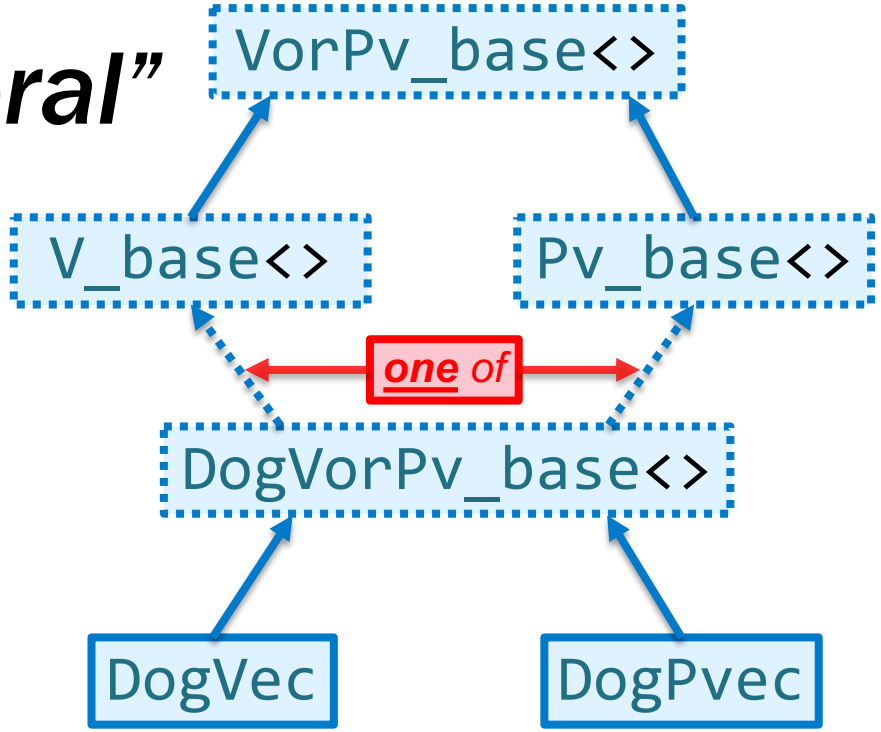
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();
```



Subsetting Our Dogs: *"Oldest Several"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
```

*Ask **vec** for oldest*

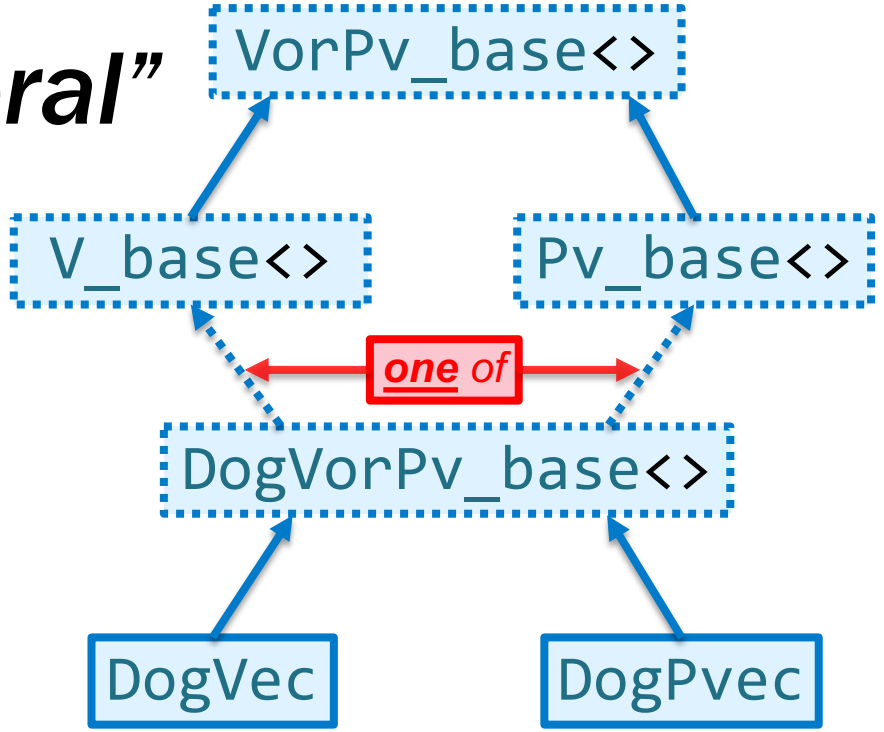


Subsetting Our Dogs: *"Oldest Several"*

```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
DogPvec dogs_oldest1 = dog_pvec.get_oldest();
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*



Subsetting Our Dogs: *"Oldest Several"*

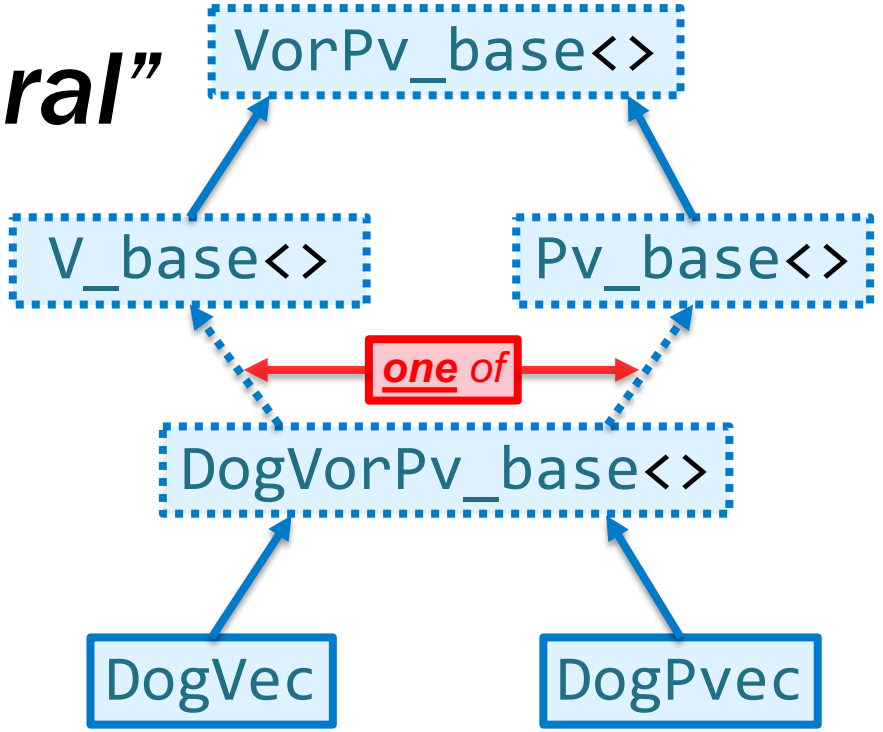
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
DogPvec dogs_oldest1 = dog_pvec.get_oldest();

PrintDogs(dog_vec);
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```



Subsetting Our Dogs: *"Oldest Several"*

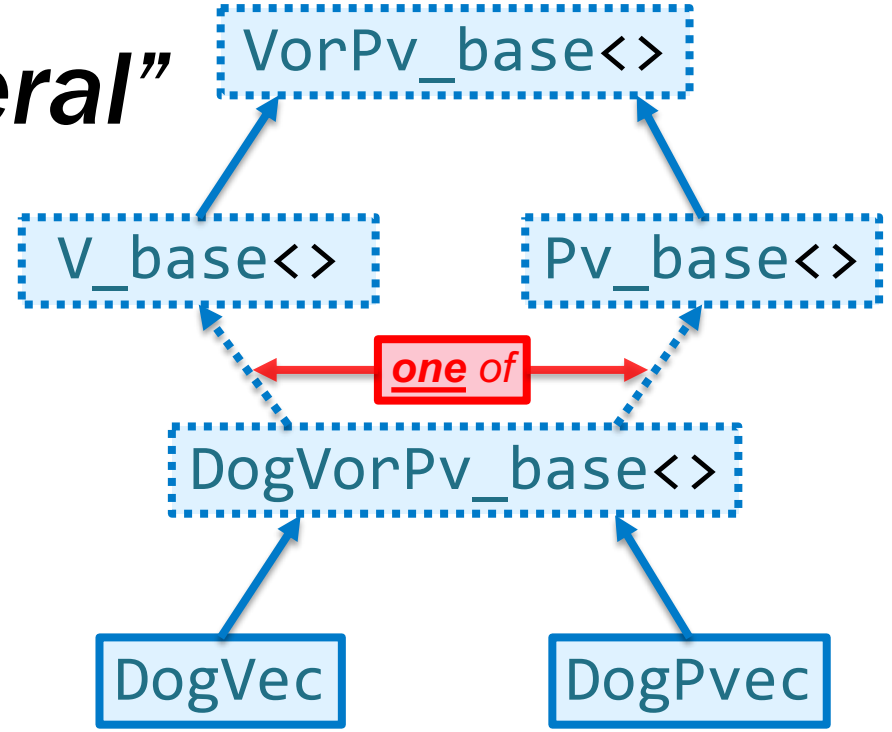
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
DogPvec dogs_oldest1 = dog_pvec.get_oldest();

PrintDogs(dog_vec);
puts("...Oldest (several):");
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
...Oldest (several):
```



Subsetting Our Dogs: *"Oldest Several"*

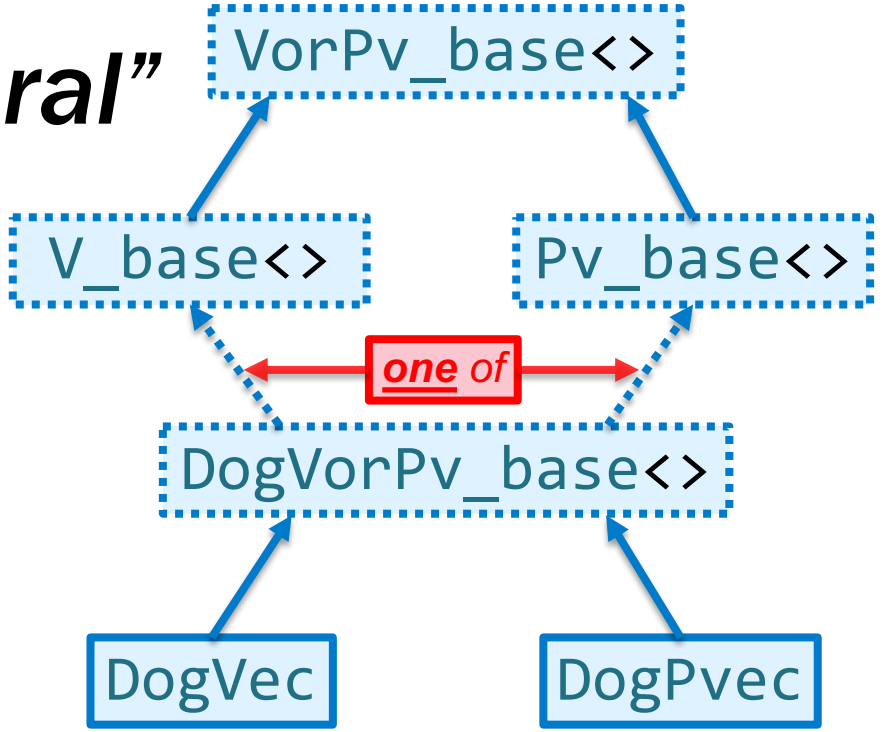
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
DogPvec dogs_oldest1 = dog_pvec.get_oldest();

PrintDogs(dog_vec);
puts("...Oldest (several):");
PrintDogs(dogs_oldest0);
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
...Oldest (several):
Bella (5 years)
Tucker (5 years)
```



Subsetting Our Dogs: *"Oldest Several"*

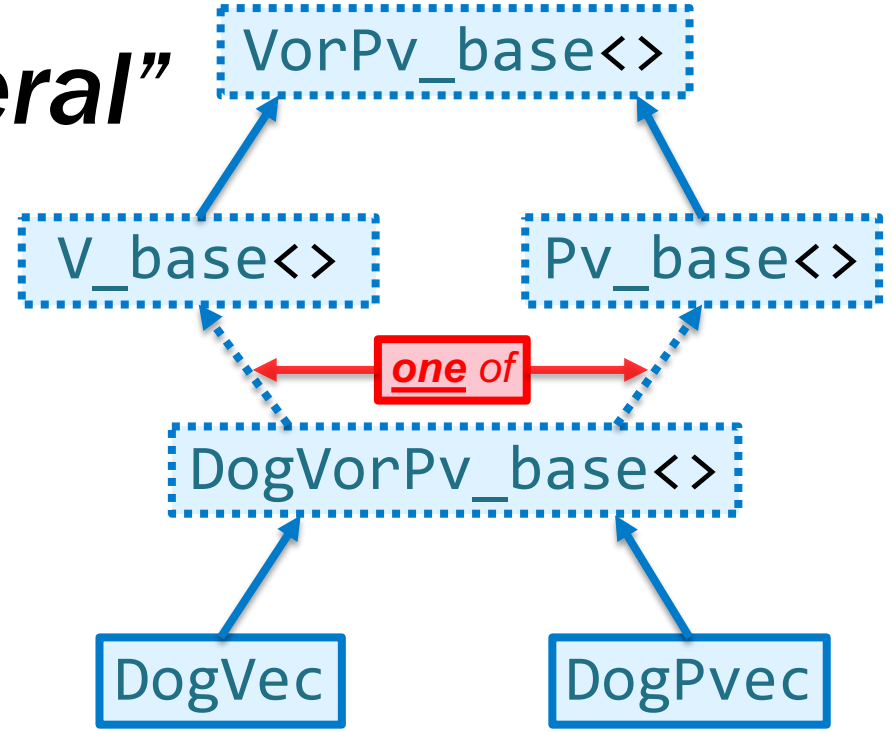
```
DogVec dog_vec;
dog_vec.addValue_native_emplace("Bella",  5_years);
dog_vec.addValue_native_emplace("Oliver", 6_months);
dog_vec.addValue_native_emplace("Lucy",  12_months);
dog_vec.addValue_native_emplace("Tucker", 5_years);

DogPvec dog_pvec = dog_vec.getAs_pvec();

DogPvec dogs_oldest0 = dog_vec.get_oldest();
DogPvec dogs_oldest1 = dog_pvec.get_oldest();

PrintDogs(dog_vec);
puts("...Oldest (several):");
PrintDogs(dogs_oldest0);
PrintDogs(dogs_oldest1);
```

*Ask **vec** for oldest*

*Ask **pvec** for oldest*

Output

```
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
...Oldest (several):
Bella (5 years)
Tucker (5 years)
Bella (5 years)
Tucker (5 years)
```



Implementation is shared
for **vec** and **pvec**

(same results produced from same algorithm)

A “Chaining” Example

- Given:

```
DogVec dog_vec = ...;
```

- Extract female dogs, sorted by age:

Declarative Algorithm:

1. Extract **`pvec`** of all female dogs
2. Sort by age

A "Chaining" Example

Declarative Algorithm:
1. Extract **`pvec`** of all female dogs
2. Sort by age

- Given:

```
DogVec dog_vec = ...;
```

- Extract female dogs, sorted by age:

Option A: **Ad-Hoc** *(i.e., "Build it as-you-need-it")*

```
DogPvec dogs_female_sorted_by_age0 =
  dog_vec.getPvec_matches_each_ref(
    [](const Dog& Dog){ return Dog.is_female(); } ).getAs_pvec_sortBy(
      [](const Dog& lhs, const Dog& rhs){ return lhs.getAge() < rhs.getAge(); } );
```

filter predicate

sort predicate

A "Chaining" Example

Declarative Algorithm:
1. Extract **`pvec`** of all female dogs
2. Sort by age

- Given:

```
DogVec dog_vec = ...;
```

- Extract female dogs, sorted by age:

Option A: **Ad-Hoc** *(i.e., "Build it as-you-need-it")*

```
DogPvec dogs_female_sorted_by_age0 =
  dog_vec.getPvec_matches_each_ref(
    [](const Dog& Dog){ return Dog.is_female(); } ).getAs_pvec_sortBy(
      [](const Dog& lhs, const Dog& rhs){ return lhs.getAge() < rhs.getAge(); } );
```

filter predicate

sort predicate

Option B: **Standardized Query** *(i.e., "reusable declarative logic")*

```
DogPvec dogs_female_sorted_by_age1 =
  dog_vec.getAsPvec_female_sortedByAge();
```

Reusable implementation in ***`DogVorPv_base<>`***

Heterogeneous Containers

Storing Disparate Types

Heterogeneous Containers

Homogeneous Container *(def):*
Objects stored have the same type

Heterogeneous Container *(def):*
Objects stored may have different types

The **`std::` containers are homogeneous**
(objects stored must have the same type)

Heterogeneous Containers

Homogeneous Container *(def):*
Objects stored have the same type

Heterogeneous Container *(def):*
Objects stored may have different types

The **`std::` containers are homogeneous**
(objects stored must have the same type)

- *Why?* **Homogenous** containers are:
  - **Most useful** *(most commonly required)*
  - **Easiest** to use *(element type is discrete)*
  - **Provide greatest safety** *(protection through the type system)*
  - Are the **most efficient** *(in "space" and "time")*
  - Give the **best compile-time errors** *(identifying improper usage)*

Heterogeneous Containers

Homogeneous Container *(def):*
Objects stored have the same type

Heterogeneous Container *(def):*
Objects stored may have different types

The **`std::` containers are homogeneous**
(objects stored must have the same type)

- *Why?* **Homogenous** containers are:
  - **Most useful** *(most commonly required)*
  - **Easiest** to use *(element type is discrete)*
  - **Provide greatest safety** *(protection through the type system)*
  - Are the **most efficient** *(in "space" and "time")*
  - Give the **best compile-time errors** *(identifying improper usage)*

A **non-homogenous** container is **heterogeneous**
(it may contain objects of different types)

Recommended Practice *(for Heterogeneous Containers)*

To store objects of **different types within a homogenous container**, you must **make a homogeneous type**
(which becomes the element type within that homogenous container)

Recommended Practice *(for Heterogeneous Containers)*

To store objects of **different types within a homogenous container**, you must **make a homogeneous type**
(which becomes the element type within that homogenous container)

- **Recommended practice for heterogeneous** containers in C++:
  - *Container of* **variants**
  - *Container of* **pointers-to-base**
  - *Container of* **shared-pointers-to-base**

```
std::vector<std::variant<Cat,Dog>>   animal_vec;
std::vector<Animal*>                 animal_vec;
std::vector<std::shared_ptr<Animal>> animal_vec;
```

Recommended Practice *(for Heterogeneous Containers)*

To store objects of **different types within a homogenous container**, you must **make a homogeneous type**
(which becomes the element type within that homogenous container)

- **Recommended practice for heterogeneous** containers in C++:
  - *Container of* **variants**
  - *Container of* **pointers-to-base**
  - *Container of* **shared-pointers-to-base**

```
std::vector<std::variant<Cat,Dog>>   animal_vec;
std::vector<Animal*>                 animal_vec;
std::vector<std::shared_ptr<Animal>> animal_vec;
```

- An ***alternative*** may be a *(**declarative**)* **Data-Oriented container**:

```
struct AnimalVec {
  CatVec cats_;
  DogVec dogs_;
};
```

```
struct AnimalPvec {
  CatPvec cats_;
  DogPvec dogs_;
};
```

Virtual-Ctor Idiom

- To **copy a container of heterogeneous types**, can use the ***"virtual constructor idiom":***
  - **Add a `virtual clone()`** member function for copy constructing
  - **Add a `virtual create()`** member function for the default constructor
  - **Relies upon covariant return types**, where the ***DERIVED*** return-type can be different from that in the ***BASE*** *(because the return type is not part of the function signature, and the* ***DERIVED*** *type is covariant with the* ***BASE*** *type)*

```
class Animal {
  public:
    virtual ~Animal() { } // virtual dtor
    virtual Animal* clone() const = 0; // Uses copy ctor
    virtual Animal* create() const = 0; // Uses default ctor
};

class Dog : public Animal {
  public:
    Dog* clone() const override  { return new Dog(*this); }
    Dog* create() const override { return new Dog();      }
};
```

clone *or* ***create*** *derived-most*

```
void FuncToCopyElement(Animal& animal) {
  Animal* animal_cloned = animal.clone();
  Animal* animal_empty  = animal.create();
  // ...
  delete animal_cloned; // virtual dtor required
  delete animal_empty;  // virtual dtor required
}
```

Annoyance: Containers Of Hierarchical Types

- *Annoyance #1:* You **don't always have a common base** class
- *Annoyance #2:* When you do have a common base class, **you can't interchange container types**
  - *Example:* You cannot assign `std::vector<Dog*>` to `std::vector<Animal*>`

From:

http://www.stroustrup.com/bs_faq2.html

Why can't I assign a vector<Apple*> to a vector<Fruit*>?

Because that would open a hole in the type system. For example:

```
class Apple : public Fruit { void apple_fct(); /* ... */ };
class Orange : public Fruit { /* ... */ }; // Orange doesn't have apple_fct()

vector<Apple*> v;       // vector of Apples

void f(vector<Fruit*>& vf)              // innocent Fruit manipulating function
{
        vf.push_back(new Orange);       // add orange to vector of fruit
}

void h()
{
        f(v);   // error: cannot pass a vector<Apple*> as a vector<Fruit*>
        for (int i=0; i<v.size(); ++i) v[i]->apple_fct();
}
```

Had the call f(v) been legal, we would have had an Orange pretending to be an Apple.

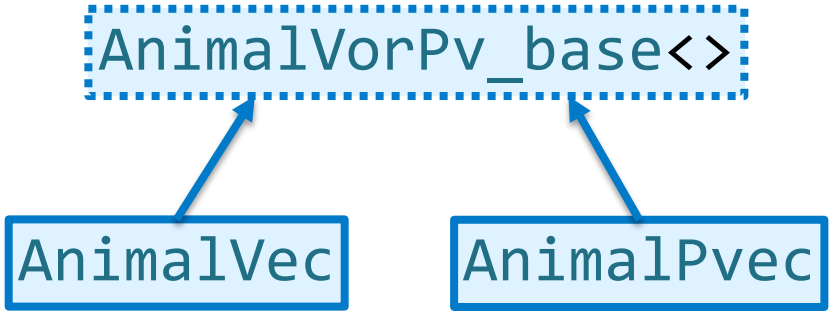
- Assuming:
  - No “common” base-class
  - No “`virtual`” at all

What might a “*Declarative*” *Heterogeneous Container* look like?

Case Study: Declarative Heterogeneous Container

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  protected:
    Cat_VorPv_ cat_vorpv_;
    Dog_VorPv_ dog_vorpv_;
  public:




};
```

Data-Oriented Layout
(of heterogeneous types)




```
class AnimalVec : public AnimalVorPv_base<AnimalVec, CatVec, DogVec> {
};
class AnimalPvec : public AnimalVorPv_base<AnimalPvec, CatPvec, DogPvec> {
};
```

Goal: Empty

insert storage spec

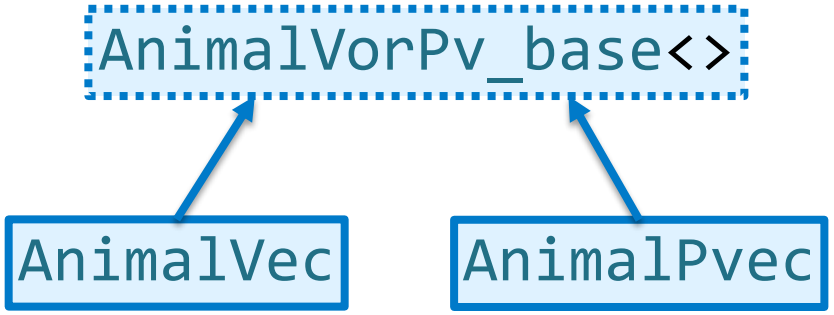
Goal: Domain logic + container manipulation

Case Study: Declarative Heterogeneous Container

```
class AnimalPvec;

template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  protected:
    Cat_VorPv_ cat_vorpv_;
    Dog_VorPv_ dog_vorpv_;
  public:
    void addValue_native(TYPE_CAT_NATIVE item_value_native);
    void addValue_native(TYPE_DOG_NATIVE item_value_native);
    ...
    template<class TYPE_PVEC>
      void appendTo_pvec(TYPE_PVEC& pvec_append_to) const;
    template<class TYPE_VEC>
      void appendTo_vec(TYPE_VEC& vec_append_to) const;

    void get_oldest(AnimalPvec& animal_pvec) const;
    AnimalPvec get_oldest(void) const;

    std::size_t getNumItems(void) const;
    ...
};
```

Data-Oriented Layout
(of heterogeneous types)

Goal: Domain logic + container manipulation



The "**heavy lifting" is done by the nested containers**.

The assembly into a heterogenous container is simple.

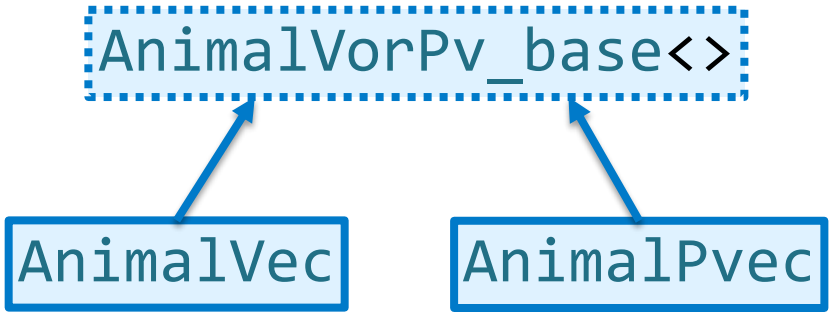
```
class AnimalVec : public AnimalVorPv_base<AnimalVec, CatVec, DogVec> {
};
class AnimalPvec : public AnimalVorPv_base<AnimalPvec, CatPvec, DogPvec> {
};
```

Goal: Empty

insert storage spec

Implementation: Heterogeneous Container

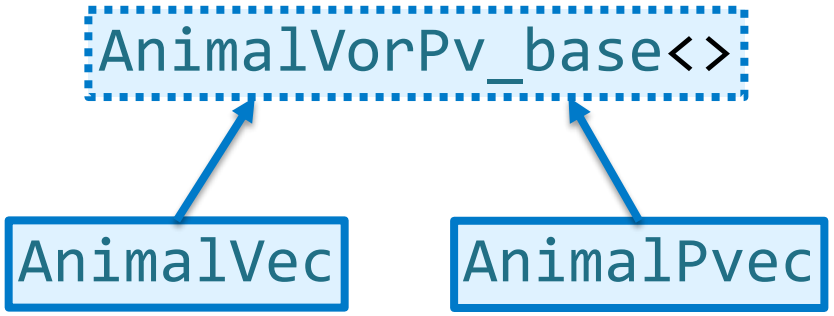
```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  public:
    void addValue_native(TYPE_CAT_NATIVE item_value_native) {
      cat_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native(TYPE_DOG_NATIVE item_value_native) {
      dog_vorpv_.addValue_native(item_value_native);
    }
```




```
};
```

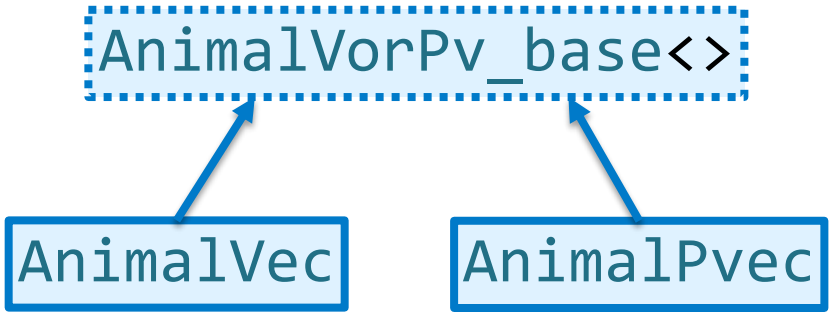
Implementation: Heterogeneous Container

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  public:
    void addValue_native(TYPE_CAT_NATIVE item_value_native) {
      cat_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native(TYPE_DOG_NATIVE item_value_native) {
      dog_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native_emplace_cat(ARGS_CTOR&&... args_ctor) {
      cat_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
    void addValue_native_emplace_dog(ARGS_CTOR&&... args_ctor) {
      dog_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }




};
```



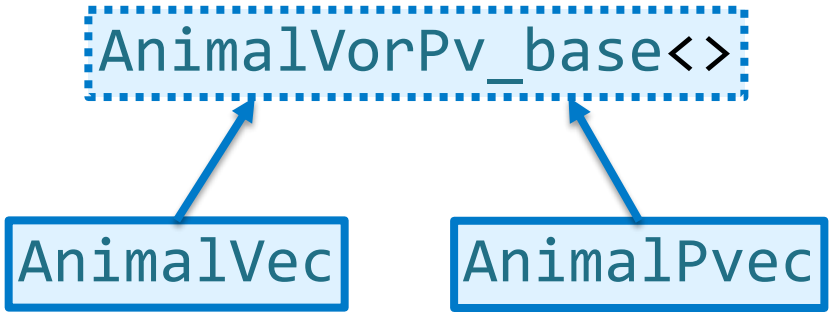
Implementation: Heterogeneous Container

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  public:
    void addValue_native(TYPE_CAT_NATIVE item_value_native) {
      cat_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native(TYPE_DOG_NATIVE item_value_native) {
      dog_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native_emplace_cat(ARGS_CTOR&&... args_ctor) {
      cat_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
    void addValue_native_emplace_dog(ARGS_CTOR&&... args_ctor) {
      dog_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
    std::size_t getNumItems(void) const {
      return cat_vorpv_.getNumItems() +
             dog_vorpv_.getNumItems();
    }
    bool hasState(void) const {
      return cat_vorpv_.hasState() ||
             dog_vorpv_.hasState();
    }
};
```



Implementation: Heterogeneous Container

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
  public:
   using TYPE_CAT_NATIVE = typename Cat_VorPv_::TYPE_ITEM_NATIVE;
   using TYPE_DOG_NATIVE = typename Dog_VorPv_::TYPE_ITEM_NATIVE;
  public:
    void addValue_native(TYPE_CAT_NATIVE item_value_native) {
      cat_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native(TYPE_DOG_NATIVE item_value_native) {
      dog_vorpv_.addValue_native(item_value_native);
    }
    void addValue_native_emplace_cat(ARGS_CTOR&&... args_ctor) {
      cat_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
    void addValue_native_emplace_dog(ARGS_CTOR&&... args_ctor) {
      dog_vorpv_.addValue_native_emplace(
        std::forward<ARGS_CTOR&&>(args_ctor)...);
    }
    std::size_t getNumItems(void) const {
      return cat_vorpv_.getNumItems() +
             dog_vorpv_.getNumItems();
    }
    bool hasState(void) const {
      return cat_vorpv_.hasState() ||
             dog_vorpv_.hasState();
    }
};
```

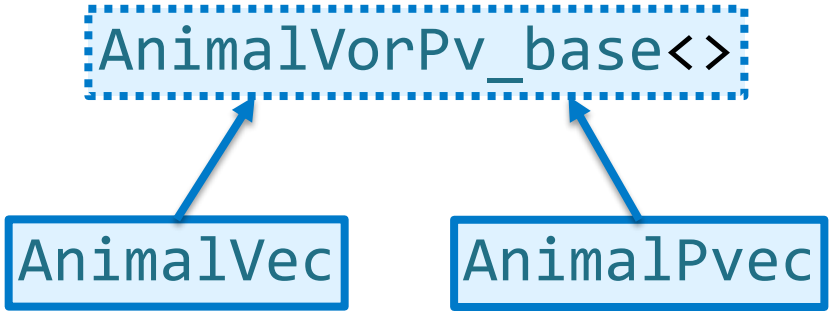


The **interface is "the same"** as that provide by the *(nested)* containers

Most **implementation is trivial**

Get As... { *vec* | *pvec* }

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& pvec_append_to) const {
      cat_vorpv_.appendTo_pvec(pvec_append_to.get_Cat_VorPv_ref());
      dog_vorpv_.appendTo_pvec(pvec_append_to.get_Dog_VorPv_ref());
    }

    template<class TYPE_PVEC>
      void getAs_pvec(TYPE_PVEC& as_pvec) const;
    template<class TYPE_PVEC>
      TYPE_PVEC getAs_pvec(void) const;

};
```

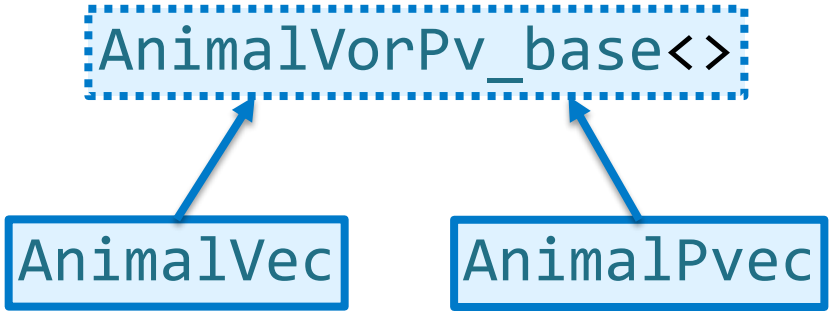


Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "{**pvec|vec**}"…

Get As... { *vec* | *pvec* }

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& pvec_append_to) const {
      cat_vorpv_.appendTo_pvec(pvec_append_to.get_Cat_VorPv_ref());
      dog_vorpv_.appendTo_pvec(pvec_append_to.get_Dog_VorPv_ref());
    }
    template<class TYPE_VEC>
    void appendTo_vec(TYPE_VEC& vec_append_to) const {
      cat_vorpv_.appendTo_vec(vec_append_to.get_Cat_VorPv_ref());
      dog_vorpv_.appendTo_vec(vec_append_to.get_Dog_VorPv_ref());
    }
    template<class TYPE_PVEC>
      void getAs_pvec(TYPE_PVEC& as_pvec) const;
    template<class TYPE_PVEC>
      TYPE_PVEC getAs_pvec(void) const;

};
```

```
template<class TYPE_VEC>
  void getAs_vec(TYPE_VEC& as_vec) const;
template<class TYPE_VEC>
  TYPE_VEC getAs_vec(void) const;
```

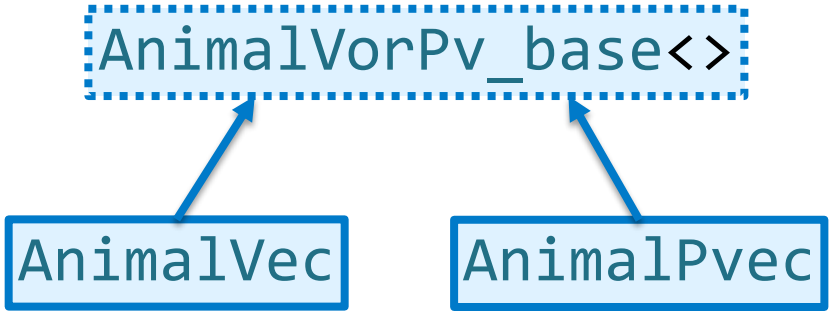


Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "{**pvec|vec**}"...

Get As... {vec|pvec} + "Short-Circuit"

```
template<class DERIVED, class Cat_VorPv_, class Dog_VorPv_>
class AnimalVorPv_base {
    ...
  public:
    template<class TYPE_PVEC>
    void appendTo_pvec(TYPE_PVEC& pvec_append_to) const {
      cat_vorpv_.appendTo_pvec(pvec_append_to.get_Cat_VorPv_ref());
      dog_vorpv_.appendTo_pvec(pvec_append_to.get_Dog_VorPv_ref());
    }
    template<class TYPE_VEC>
    void appendTo_vec(TYPE_VEC& vec_append_to) const {
      cat_vorpv_.appendTo_vec(vec_append_to.get_Cat_VorPv_ref());
      dog_vorpv_.appendTo_vec(vec_append_to.get_Dog_VorPv_ref());
    }
    template<class TYPE_PVEC>
      void getAs_pvec(TYPE_PVEC& as_pvec) const;
    template<class TYPE_PVEC>
      TYPE_PVEC getAs_pvec(void) const;

};
```

```
template<class TYPE_VEC>
  void getAs_vec(TYPE_VEC& as_vec) const;
template<class TYPE_VEC>
  TYPE_VEC getAs_vec(void) const;
```



Goal: Whatever our "container", we ***"get"*** its state ***"as-if"*** it were a "{**pvec|vec**}"…

Note: If our container ***is*** that type, merely return ****this***

```
class AnimalVec : public AnimalVorPv_base<AnimalVec, CatVec, DogVec>
{
  using BASE =  AnimalVorPv_base<AnimalVec, CatVec, DogVec>;
  public:
    using BASE::getAs_vec;
    const AnimalVec& getAs_vec(void) const {
      return *this;
    }
};
```

"Do Nothing" optimization

```
class AnimalPvec : public AnimalVorPv_base<AnimalPvec, CatPvec, DogPvec>
{
  using BASE =  AnimalVorPv_base<AnimalPvec, CatPvec, DogPvec>;
  public:
    using BASE::getAs_pvec;
    const AnimalPvec& getAs_pvec(void) const {
      return *this;
    }
};
```

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
```

Top 10 Cat Names

of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
```

Top 10 Cat Names

of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
```

Top 10 Cat Names

of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
```

Top 10 Cat Names

of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));

animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));
```

Top 10 Cat Names

of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));

animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

PrintAnimals(animal_vec);
```

Output

```
Luna (5 years)
Leo (3 months)
Milo (6 months)
Nala (5 months)
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```

Top 10 Cat Names
of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));

animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

PrintAnimals(animal_vec);
```

Output

```
Luna (5 years)
Leo (3 months)
Milo (6 months)
Nala (5 months)
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```

Default implementation prints "Cats", then "Dogs" (this is changed in application-code with filters and iterators)

Top 10 Cat Names
of 2018

Male	Female
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));

animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

PrintAnimals(animal_vec);


AnimalPvec animal_pvec = animal_vec.getAs_pvec();
```

Output

```
Luna (5 years)
Leo (3 months)
Milo (6 months)
Nala (5 months)
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```

Default implementation prints "Cats", then "Dogs" (this is changed in application-code with filters and iterators)

Top 10 Cat Names
of 2018

Male	Female
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

Playing With Our Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));

animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));

animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));

animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

PrintAnimals(animal_vec);


AnimalPvec animal_pvec = animal_vec.getAs_pvec();

PrintAnimals(animal_pvec);
```

Output *(x2)*

```
Luna (5 years)
Leo (3 months)
Milo (6 months)
Nala (5 months)
Bella (5 years)
Oliver (6 months)
Lucy (12 months)
Tucker (5 years)
```

Default implementation prints "Cats", then "Dogs" (this is changed in application-code with filters and iterators)

Top 10 Cat Names
of 2018

Male	***Female***
Oliver	Luna
Leo	Chloe
Charlie	Bella
Milo	Lucy
Max	Lily
Jack	Sophie
George	Lola
Simon	Zoe
Loki	Cleo
Simba	Nala

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));
```

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();
```

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec’)
```

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec’)
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
```

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)
```

*“Do Nothing” optimization (returns **`*this`**)*

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)



const AnimalVec& animal_vec_0 = animal_pvec.getAs_vec();     // ok (reference lifetime extension)
```

*“Do Nothing” optimization (returns **`*this`**)*

“Get As” Cats And Dogs

```
AnalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)
```

“Do Nothing” optimization (returns `*this`)

```
const AnimalVec& animal_vec_0 = animal_pvec.getAs_vec();     // ok (reference lifetime extension)
const AnimalVec& animal_vec_1 = animal_vec.getAs_vec();      // ok (reference to 'animal_vec')
```

“Get As” Cats And Dogs

```
AnalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)
```

“Do Nothing” optimization (returns `*this`)

```
const AnimalVec& animal_vec_0 = animal_pvec.getAs_vec();     // ok (reference lifetime extension)
const AnimalVec& animal_vec_1 = animal_vec.getAs_vec();      // ok (reference to 'animal_vec')
assert(&animal_vec_1 == &animal_vec);                        // ok (same instance)
```

“Do Nothing” optimization (returns `*this`)

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)
```

*“Do Nothing” optimization (returns **`*this`**)*

```
const AnimalVec& animal_vec_0 = animal_pvec.getAs_vec();     // ok (reference lifetime extension)
const AnimalVec& animal_vec_1 = animal_vec.getAs_vec();      // ok (reference to 'animal_vec')
assert(&animal_vec_1 == &animal_vec);                        // ok (same instance)
```

*“Do Nothing” optimization (returns **`*this`**)*

```
AnimalPvec animal_pvec_oldest = animal_vec.get_oldest();
```

“Get As” Cats And Dogs

```
AnimalVec animal_vec;
animal_vec.addValue_native(Dog("Bella", 5_years));
animal_vec.addValue_native(Dog("Oliver", 6_months));
animal_vec.addValue_native(Cat("Luna", 5_years));
animal_vec.addValue_native(Cat("Leo", 3_months));
animal_vec.addValue_native(Dog("Lucy", 12_months));
animal_vec.addValue_native(Dog("Tucker", 5_years));
animal_vec.addValue_native(Cat("Milo", 6_months));
animal_vec.addValue_native(Cat("Nala", 5_months));

AnimalPvec animal_pvec = animal_vec.getAs_pvec();

const AnimalPvec& animal_pvec_0  = animal_pvec.getAs_pvec(); // ok (reference to 'animal_pvec')
const AnimalPvec& animal_pvec_1  = animal_vec.getAs_pvec();  // ok (reference lifetime extension)
assert(&animal_pvec_0 == &animal_pvec);                      // ok (same instance)
```

*“Do Nothing” optimization (returns ***this**)*

```
const AnimalVec& animal_vec_0 = animal_pvec.getAs_vec();     // ok (reference lifetime extension)
const AnimalVec& animal_vec_1 = animal_vec.getAs_vec();      // ok (reference to 'animal_vec')
assert(&animal_vec_1 == &animal_vec);                        // ok (same instance)
```

*“Do Nothing” optimization (returns ***this**)*

```
AnimalPvec animal_pvec_oldest = animal_vec.get_oldest();
PrintAnimals(animal_pvec_oldest);
```

Output

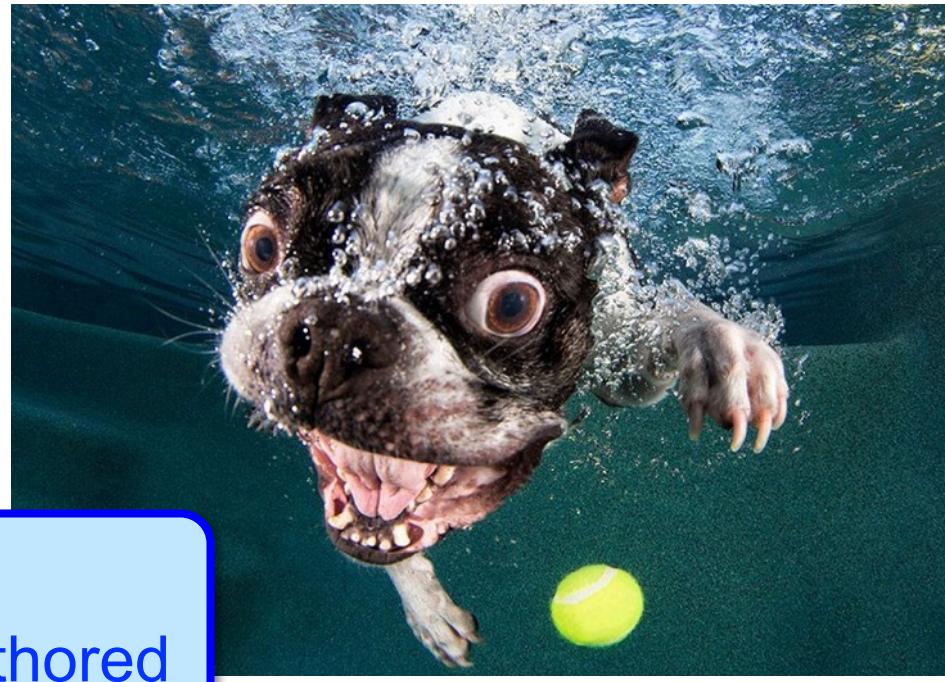
```
Luna (5 years)
Bella (5 years)
Tucker (5 years)
```

Next Steps: Heterogeneous Container

- Assemble **Higher-Order Declarative Containers** *(examples):*
  - A "**database**" is *implicitly* **a container of heterogeneous types**
  - **Application-specific data structures** *(such as "`Configuration`")* are commonly heterogeneous containers

Next Steps: Heterogeneous Container

- Assemble **Higher-Order Declarative Containers** *(examples):*
  - A "**database**" is *implicitly* **a container of heterogeneous types**
  - **Application-specific data structures** *(such as "`Configuration`")* are commonly heterogeneous containers



- **Custom Iterators** *(examples):*
  - Every-other-item toggles between `Cat` or `Dog`
  - Insertion order is "remembered" *(preserved)*

Custom Iterators Allow:

- Application-specific imperative algorithms to be authored
- `std::` algorithms to be used directly

Declarative Container Techniques

Using Your Abstraction

Use Declarative Containers In APIs!

A “Declarative” Container **attempts to be “non-leaky”**

- **Exposes Functional** Requirements
- **Does Not Expose Non-Functional** Requirements *(as much as is practical)*

Use Declarative Containers In APIs!

A "Declarative" Container **attempts to be "non-leaky"**
- **Exposes Functional** Requirements
- **Does Not Expose Non-Functional** Requirements *(as much as is practical)*

Non-Leaky Containers **can be more "stable"** *(both API and ABI)*

- Declarative Containers **tend to be "more suitable" for use in APIs**
  - **Implementation may evolve** over time *(independent of the interface)*
  - **Interface is more "flexible"**
    - Relies upon property extraction, type conversion, type coercion
    - Interface may deviate from "iterator" requirements *(e.g., iterator algorithms require availability of `end()` sentinel)*
  - **Interface is defined in terms of domain-appropriate types** *(functional behavior is more obvious)*
- Declarative Containers provide centralized management to **aggregate domain logic** used to manipulate collections of domain objects

Techniques Encouraged *with Declarative Containers*

- **Use In APIs:** Non-leaky containers tend to be more suitable *(and stable)* for use in APIs
- **Heterogeneous Containers:** Hidden storage presents simplified interface over disparate types
- **Slicing, Subsetting, and Data Reduction:** Extracting properties from ordering and filtering internal state
- **Type Conversion:** Hidden implementation can "present" internal state "as if" it is a different type *(i.e., "`getAs()`")*
- **Type Coercion:** Extracted properties may be "coerced" or be convertible to local-context types *(powerful for data cleaning and data transforms)*

Conclusion

You’re already doing Declarative,

But you should do more

Do ***NOT***

Type-Erase your domain to `std::vector<int>`

Instead,

Leverage The Type System!

```
struct HistogramData
{
    std::vector<int> bins_;
};
```

Observation

- These are different:

1. **Domain-Specific Abstraction**
   Used to compose higher-order components
   that maintain business logic invariants

2. **Computer Science Artifact**
   Used to compose higher-order components
   with anticipated technical behavior

Observation Elaboration

The C++ `std::` libraries

tend to be **good artifacts** for Computer Science

(compositional abstractions with well-understood behavioral guarantees),

but tend to make **poor domain-specific APIs**

(where an application API must necessarily be defined
in terms of key abstractions, domain-specific invariants,
and opinionated "source➔sink" lifecycles).

Observation Elaboration

The C++ `std::` libraries

tend to be **good artifacts** for Computer Science

(compositional abstractions with well-understood behavioral guarantees),

but tend to make **poor domain-specific APIs**

(where an application API must necessarily be defined in terms of key abstractions, domain-specific invariants, and opinionated "source➔sink" lifecycles).

Summarized:

1. The C++ `std::` libraries are **necessary implementation artifacts**
2. Domain-specific containers are **necessary design components**

Contrasting Declarative vs. Imperative

- **Summary Observations** *(contrasting Declarative vs. Imperative):*

Declarative:
You give up control *(over "How")* for simplified expression of common use cases

*"The **80** in the 80/20 rule"*

***Why Not**?*
*You might need greater control over "How", like to (efficiently) trigger **multiple side-effects** or compute **multiple by-products** from a **single** loop iteration*

Imperative:
You retain fine-grained control-flow to efficiently couple with localized context

*"The **20** in the 80/20 rule"*

***Why Not**?*
*You could instead write **simpler expressions** with **less code** that **avoids** many types of **errors and edge cases***

Summary: Declarative Containers

1

Declarative Containers have meaning greater than the *"sum of the parts"*

- Local types are defined
- Local semantics are implied
- Domain-specific invariants are enforced

Summary: Declarative Containers

1. **Declarative Containers have meaning greater than the *"sum of the parts"***
   - Local types are defined
   - Local semantics are implied
   - Domain-specific invariants are enforced

2. **"Declarative" implies *"Logical"***
   - `std::` containers are "physical" containers with physical properties *(like* `size` *and* `reserved_size`*)*
   - Declarative containers are "logical" with logical properties *(like* `max_age` *and* `percent_female`*)*

Summary: Declarative Containers

1. **Declarative Containers have meaning greater than the *"sum of the parts"***
   - Local types are defined
   - Local semantics are implied
   - Domain-specific invariants are enforced
2. **"Declarative" implies *"Logical"***
   - `std::` containers are "physical" containers with physical properties *(like `size` and `reserved_size`)*
   - Declarative containers are "logical" with logical properties *(like `max_age` and `percent_female`)*
3. **Declarative Containers** *(attempt to)* **provide a non-leaky abstraction**
   - Domain-specific properties may be extracted
   - Implementation may change over time

Summary: Declarative Containers

1. **Declarative Containers have meaning greater than the *"sum of the parts"***
   - Local types are defined
   - Local semantics are implied
   - Domain-specific invariants are enforced
2. **"Declarative" implies *"Logical"***
   - `std::` containers are "physical" containers with physical properties *(like `size` and `reserved_size`)*
   - Declarative containers are "logical" with logical properties *(like `max_age` and `percent_female`)*
3. **Declarative Containers** *(attempt to)* **provide a non-leaky abstraction**
   - Domain-specific properties may be extracted
   - Implementation may change over time
4. **The "Best" Container does not exist**
   - A given domain-specific type will likely participate in different containers due to constraints imposed from different algorithms
   - Thus, it is expected practice to compose your own containers for your domain-specific purposes *(this is encouraged by the C++ Standard)*

Colorado++

https://coloradoplusplus.info/

Thank you for coming!