# Value Semantics

## *It ain't about the syntax!*

John Lakos

Friday, May 16, 2014

# Copyright Notice

# Abstract

When people talk about a type as having *value* *semantics*, they are often thinking about its ability to be passed to (or returned from) a function by value. In order to do that, the C++ language requires that the type implement a copy constructor, and so people routinely implement copy constructors on their classes, which begs the question, "Should an object of that type be copyable at all?" If so, what should be true about the copy? Should it have the same state as the original object? Same behavior? What does copying an object mean?!

By *value* *type*, most people assume that the type is specifically intended to represent a member of some set (of values). A value-semantic type, however, is one that strives to approximate an abstract *mathematical* type (e.g., integer, character set, complex-number sequence), which comprises operations as well as values. When we copy an object of a value-semantic type, the new object might not have the same state, or even the same behavior as the original object; for proper value semantic types, however, the new object will have the same value.

In this talk, we begin by gaining an intuitive feel for what we mean by *value* by identifying *salient* *attributes*, i.e., those that contribute to value, and by contrasting types whose objects naturally represent values with those that don't. After quickly reviewing the syntactic properties common to typical value types, we dive into the much deeper issues that value semantics entail. In particular, we explore the subtle Essential Property of Value, which applies to every *salient* mutating operation on a value-semantic object, and then profitably apply this property to realize a correct design for each of a variety of increasingly interesting (value-semantic) classes.

# Outline

1. Introduction and Background

   Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

   Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

   Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

   What must be remembered when designing value types

# Outline

1. Introduction and Background

   Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

   Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

   Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

   What must be remembered when designing value types

1. Introduction and Background

# What's the Problem?

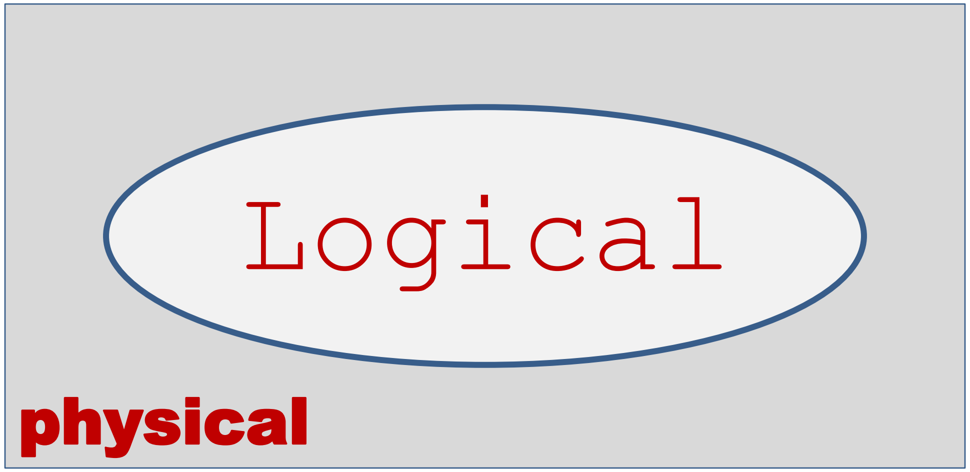
# What's the Problem?

Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.

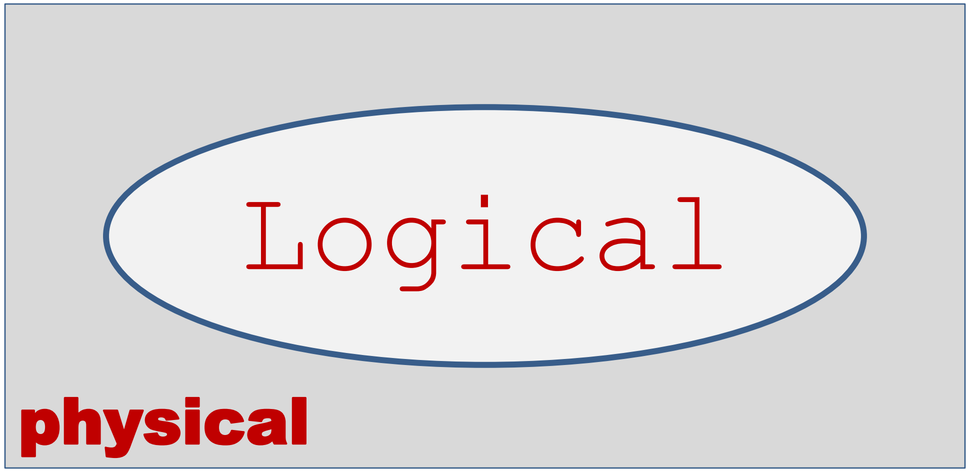
# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



# Logical versus Physical Design

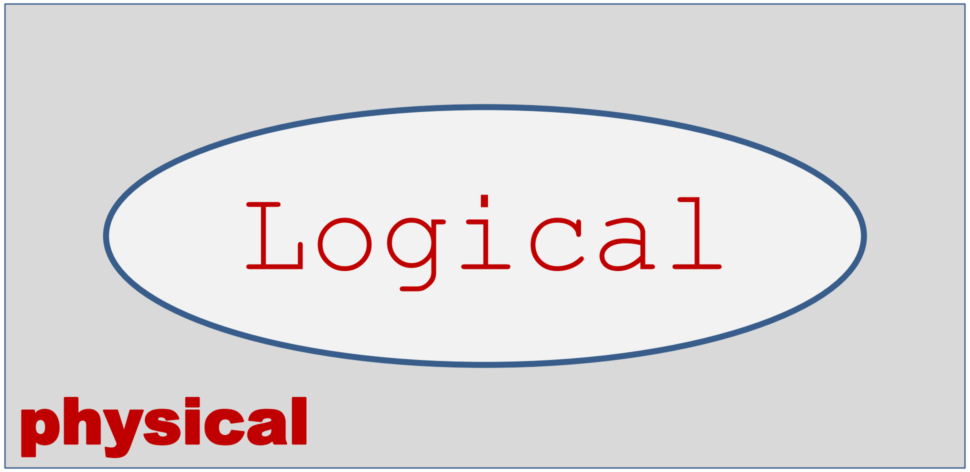
What distinguishes *Logical* from *Physical* Design?




Logical: Classes and Functions

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

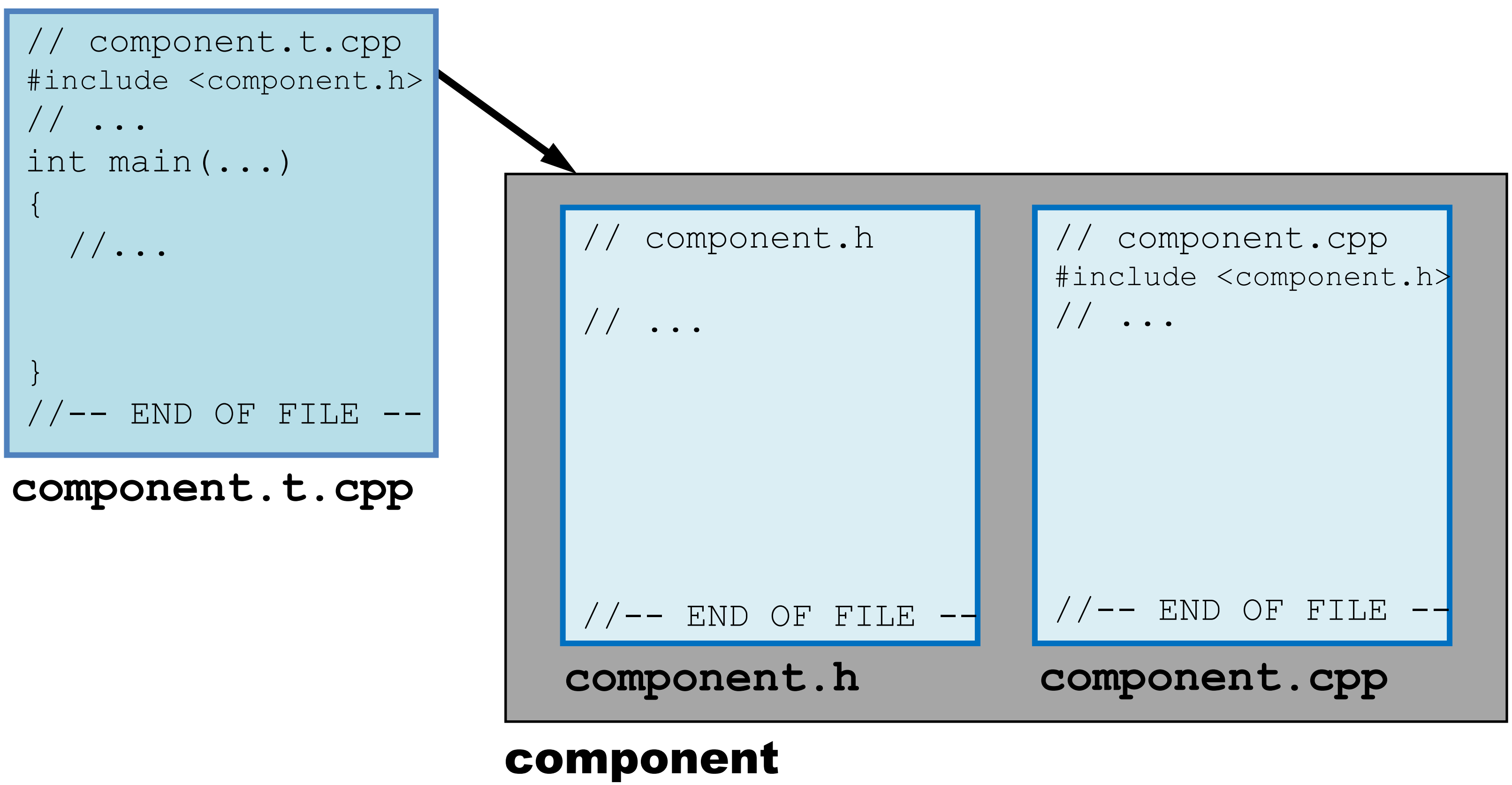



Logical: Classes and Functions

Physical: Files and Libraries

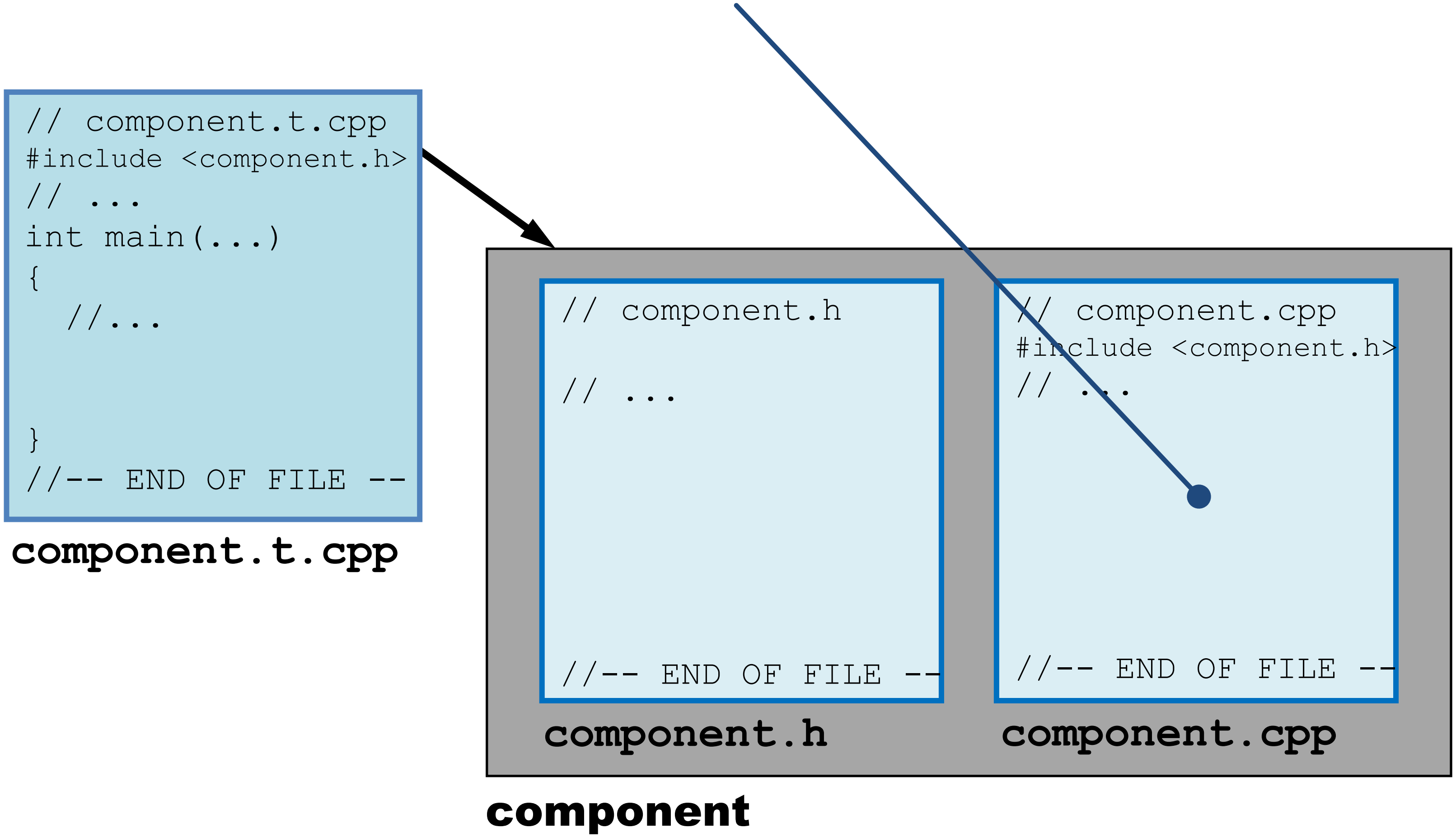
# *Component:* Uniform Physical Structure

## A Component Is Physical

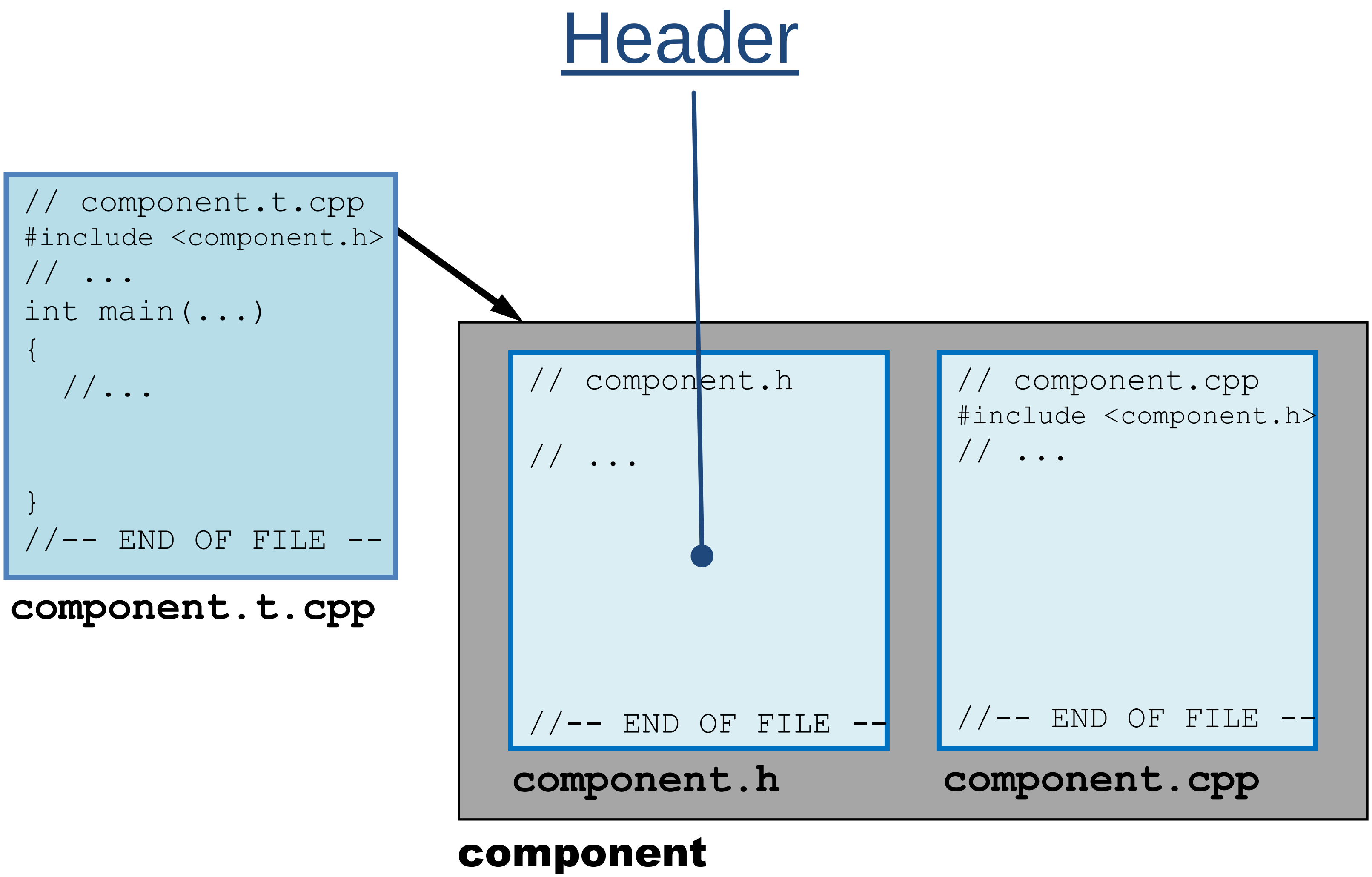


# *Component:* Uniform Physical Structure

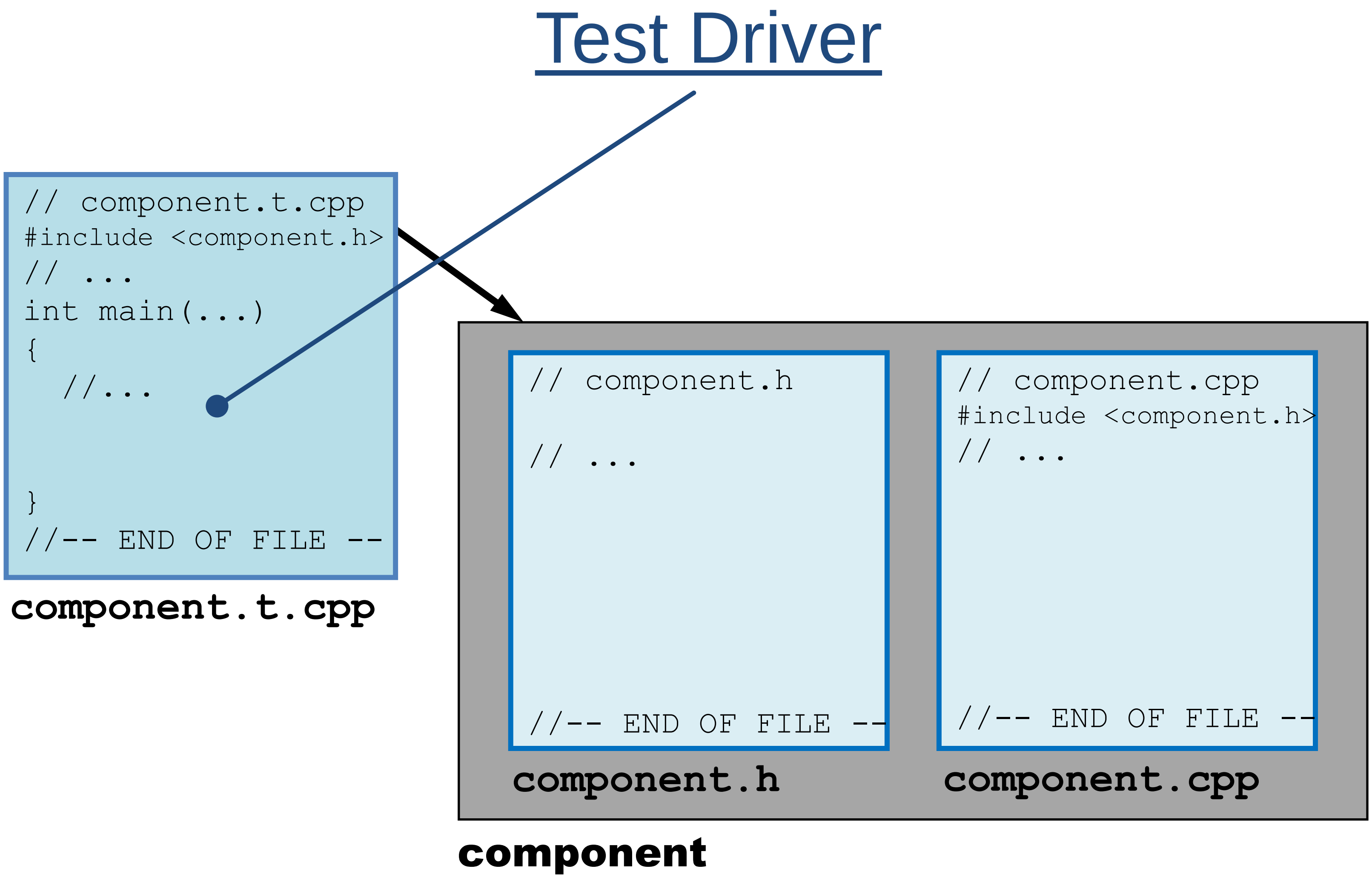
## Implementation



# *Component:* Uniform Physical Structure

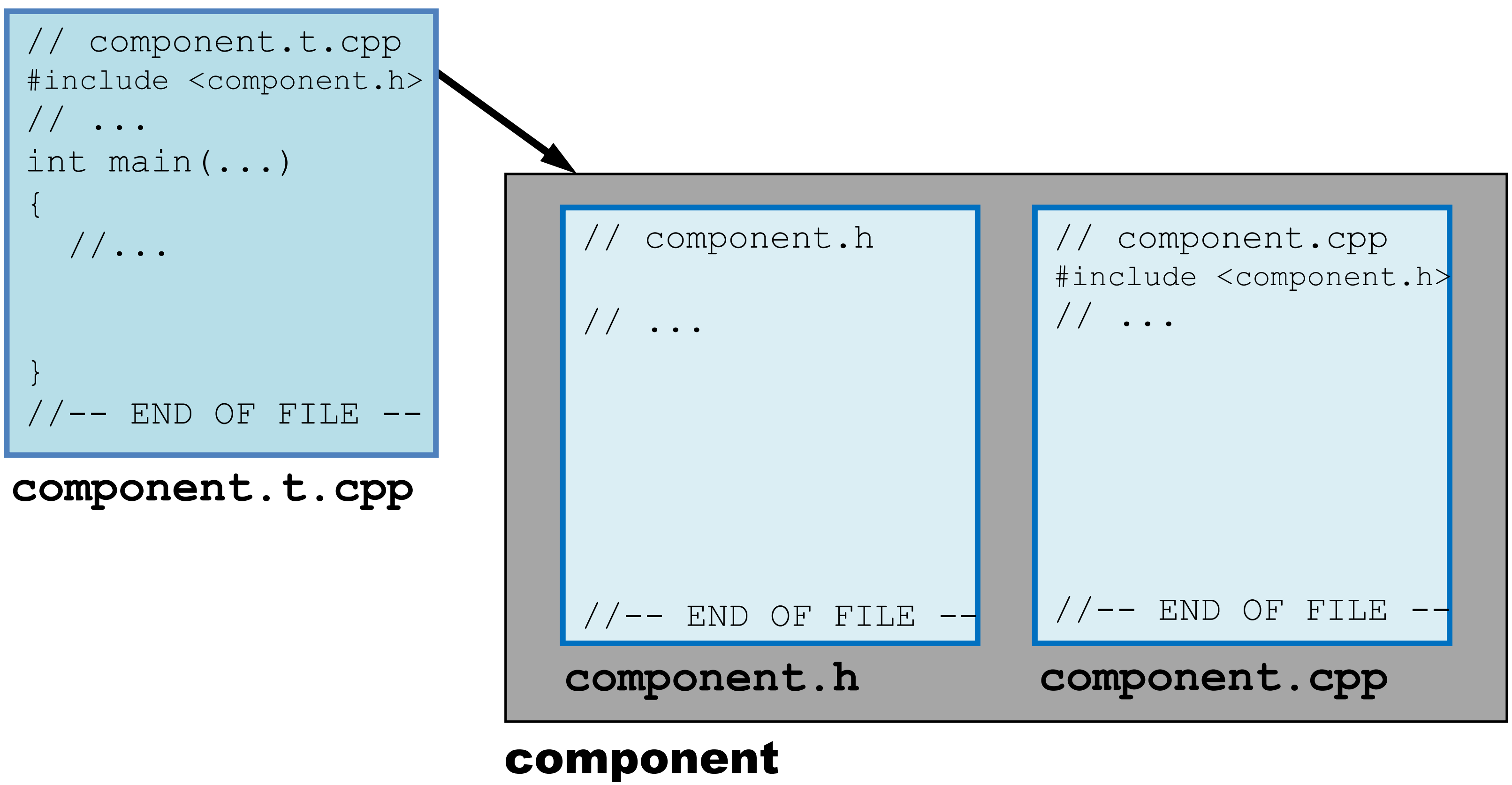


# *Component:* Uniform Physical Structure



# *Component:* Uniform Physical Structure

## <u>The</u> Fundamental Unit of Design



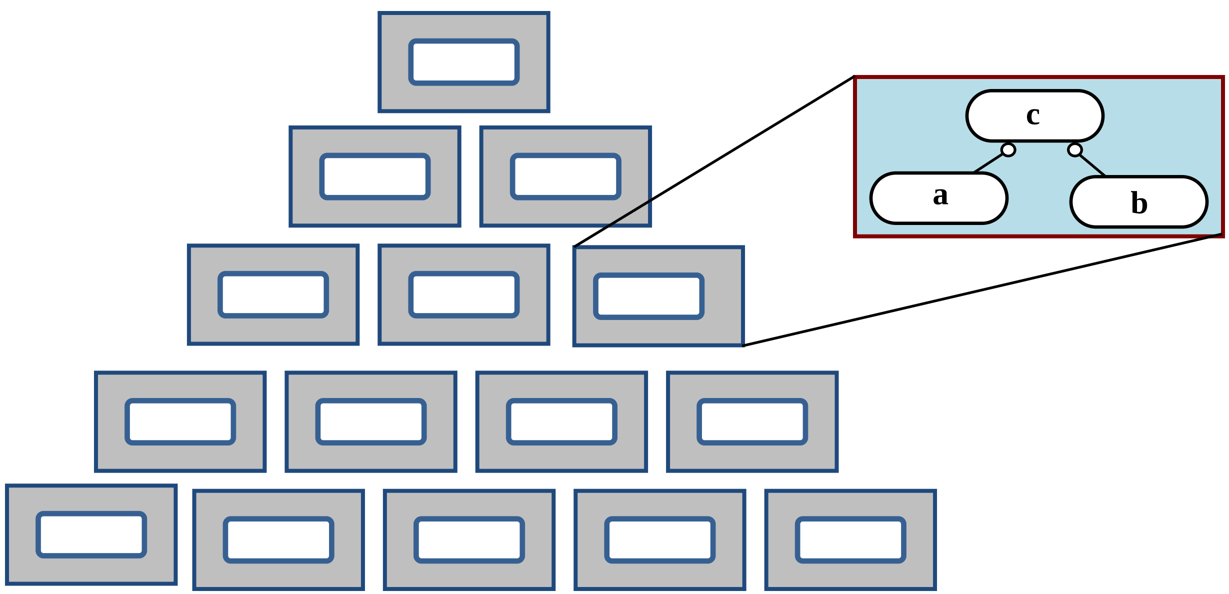
# What's the Problem?

Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.

# Logical versus Physical Design

*Logical* content aggregated into a *Physical* hierarchy of ***components***

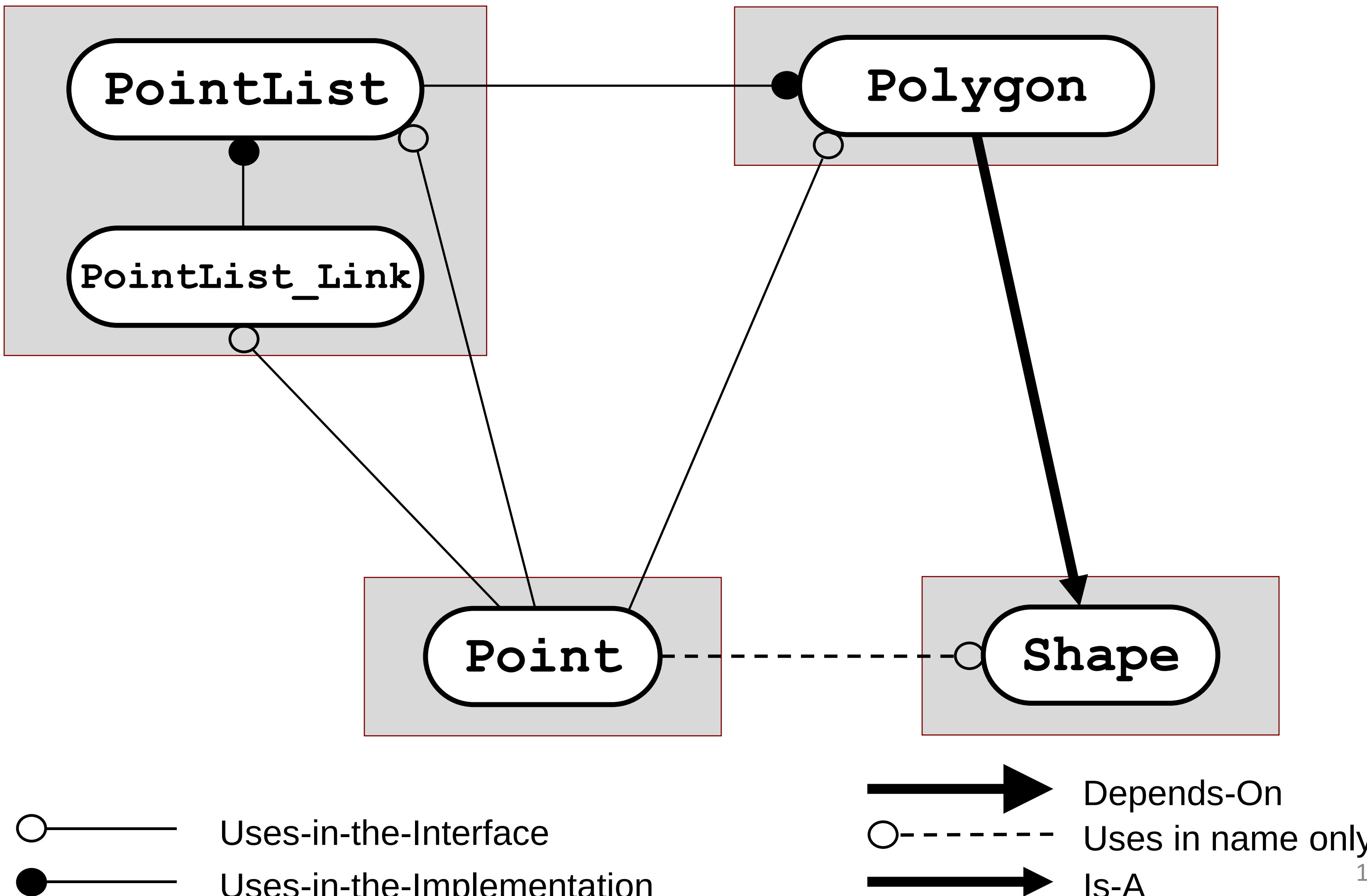


# What's the Problem?

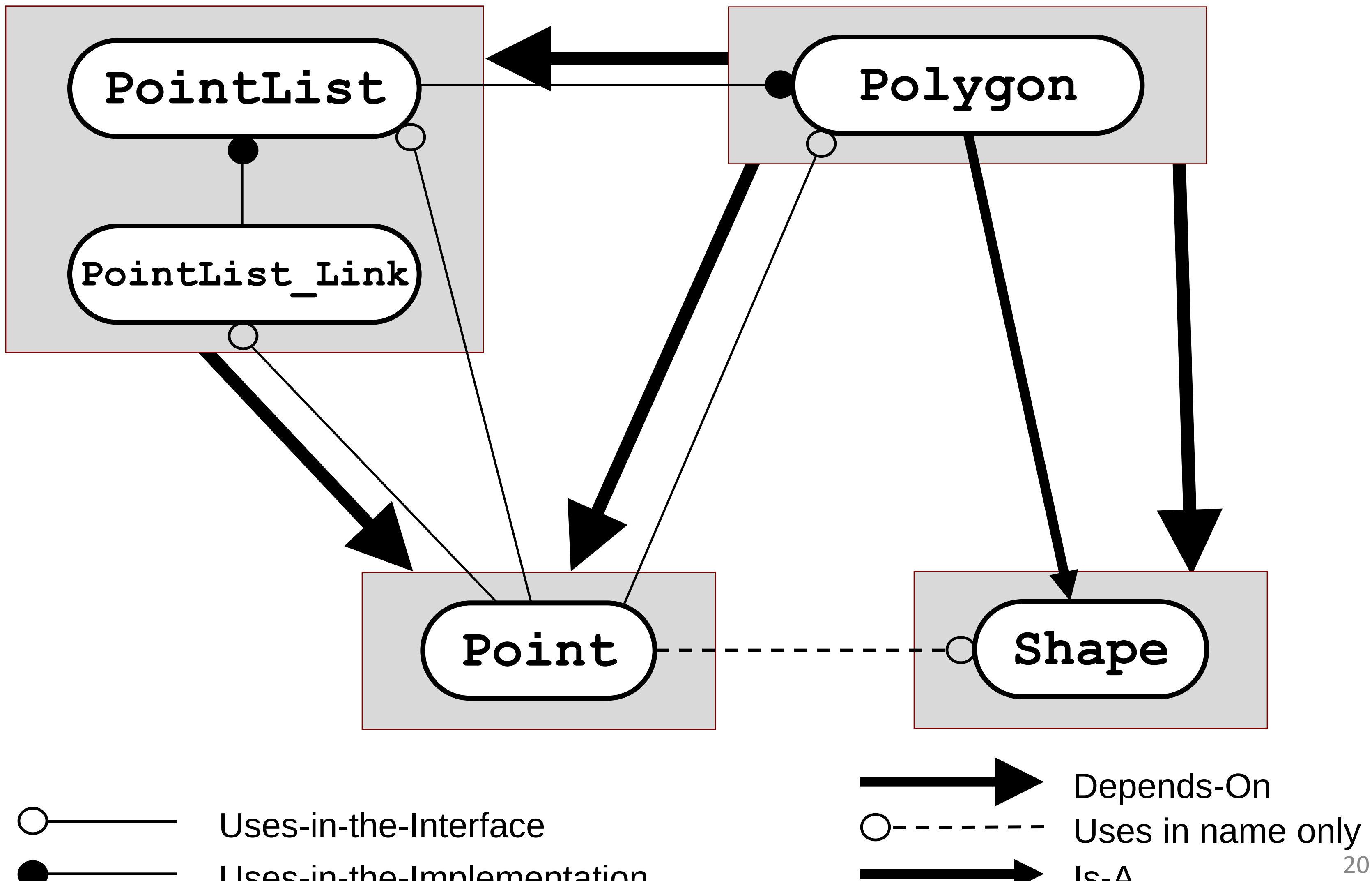
## Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- Compels the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.

# Implied Dependency



# Implied Dependency

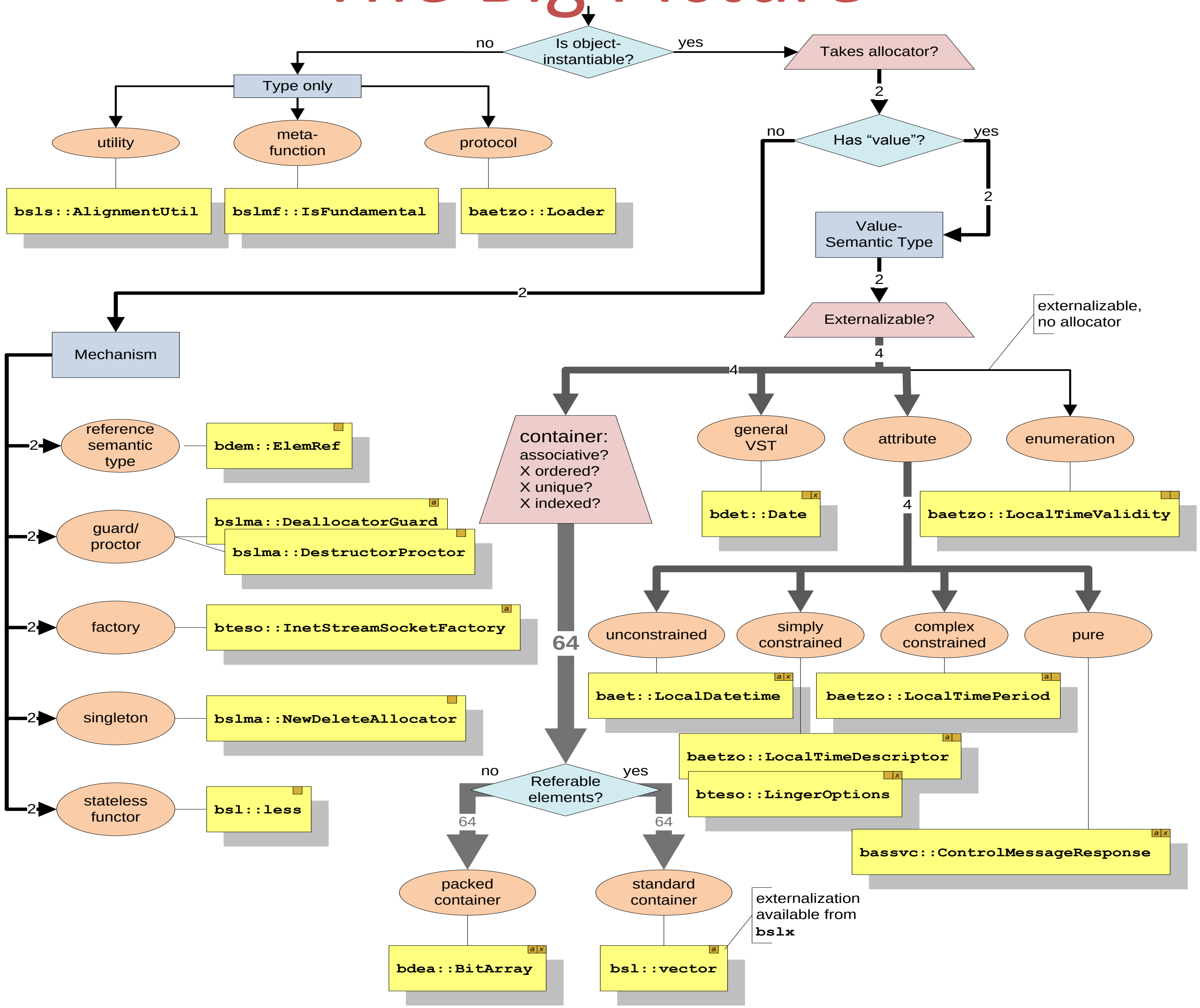


# What's the Problem?

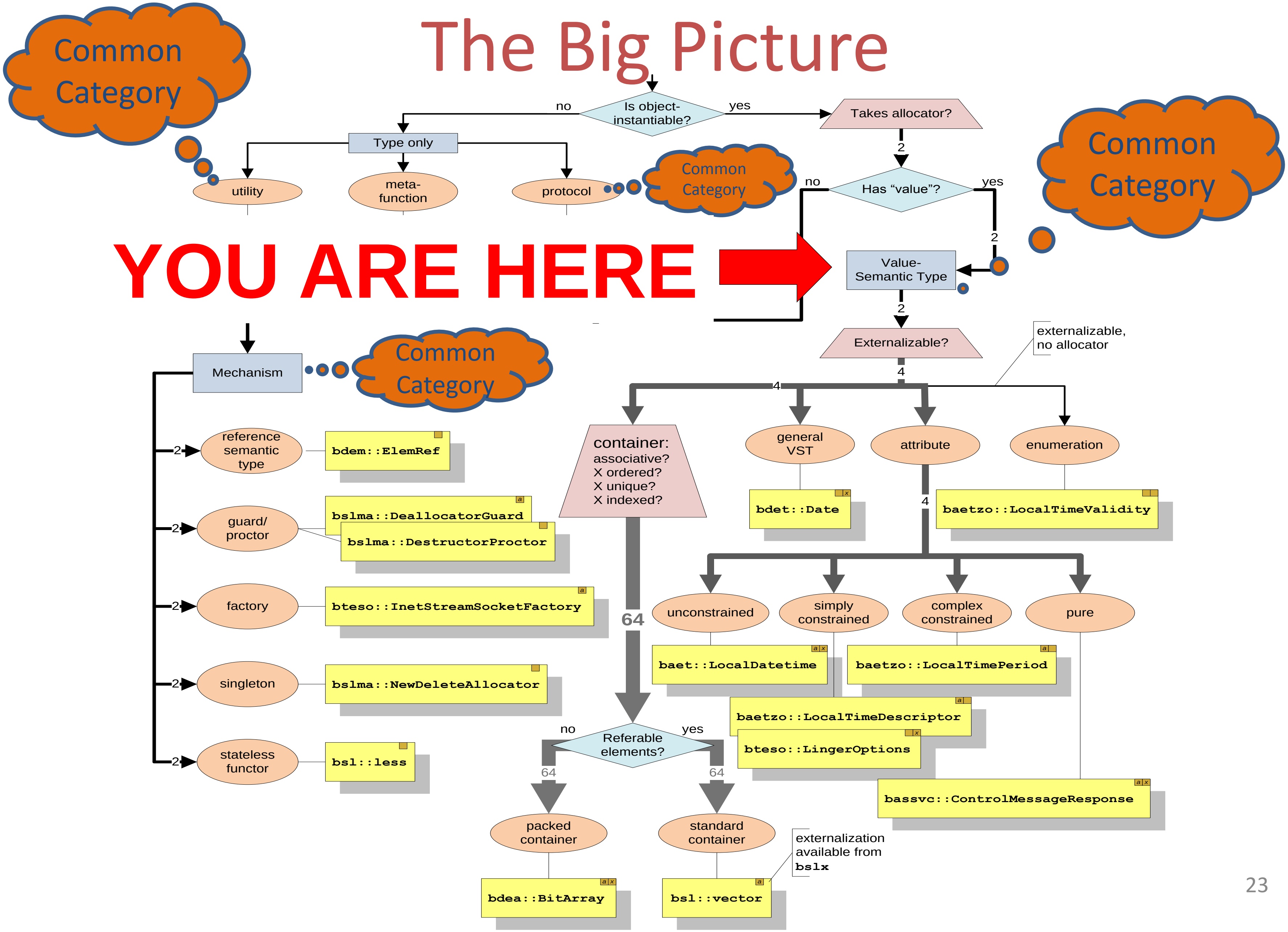
Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- Compels the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- Demands a consistent, shared understanding of the properties of common class categories: Value Types.

# The Big Picture



# The Big Picture



# Outline

1. Introduction and Background

   Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

   Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

   Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

   What must be remembered when designing value types

# Outline

1. Introduction and Background

   Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

   Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

   Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

   What must be remembered when designing value types

# Purpose of this Talk

Answer some key questions about *value*:

- What do we mean by *value*?
- Why is the notion of value important?
- Which types should be considered value types?
- What do we expect *syntactically* of a value type?
- What *semantics* should its operations have?
- How do we design proper value-semantic types?
- When should value-related syntax be omitted?

# Value versus Non-Value Types

Getting Started:

2. Understanding Value Semantics

# Value versus Non-Value Types

## Getting Started:

- Not all useful C++ classes are value types.

2. Understanding Value Semantics

# Value versus Non-Value Types

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.

# Value versus Non-Value Types

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.
- Let’s begin with understanding some basic properties of value types.

# Value versus Non-Value Types

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.
- Let’s begin with understanding some basic properties of value types.
- Then we’ll contrast them with non-value types, to create a type-category hierarchy.

# Value versus Non-Value Types

Getting Started:

- Not all useful C++ classes are value types.
- Still, value types form an important category.
- Let’s begin with understanding some basic properties of value types.
- Then we’ll contrast them with non-value types, to create a type-category hierarchy.
- After that, we’ll dig further into the details of value syntax and semantics.

# True Story

- Date: Friday Morning, October 5th, 2007
- Place: LWG, Kona, Hawaii
- Defect: issue #684: Wording of Working Paper

# True Story

- Date: Friday Morning, October 5th, 2007
- Place: LWG, Kona, Hawaii
- Defect: issue #684: Wording of Working Paper

What was meant by stating that two

`std::match_result`

objects (§28.10) were "the same" ?

# “The Same”

What do we mean by “the same”?

# "The Same"

## What do we mean by "the same"?

- The two objects are *identical*?
  - same address, same process, same time?

# "The Same"

## What do we mean by "the same"?

- The two objects are *identical*?
  - same address, same process, same time?
- The two objects are *distinct*, yet have certain *properties* in common.

# "The Same"

## What do we mean by "the same"?

- The two objects are *identical*?
  - same address, same process, same time?
- The two objects are *distinct*, yet have certain *properties* in common.

(It turned out to be the latter.)

# "The Same"

## What do we mean by "the same"?

- The two objects are *identical*?
  - same address, same process, same time?
- The two objects are *distinct*, yet have certain *properties* in common.

(It turned out to be the latter.)

## So the meaning was clear…

# "The Same"

## What do we mean by "the same"?

- The two objects are *identical*?
  - same address, same process, same time?
- The two objects are *distinct*, yet have certain *properties* in common.

(It turned out to be the latter.)

So the meaning was clear… *Or was it?*

# What exactly has to be “the Same”?

The discussion continued…

…some voiced suggestions:

# What exactly has to be “the Same”?

The discussion continued…

…some voiced suggestions:

- Whatever the copy constructor preserves.

# What exactly has to be "the Same"?

The discussion continued…

…some voiced suggestions:

- Whatever the copy constructor preserves.
- As long as the two are "equal".

# What exactly has to be "the Same"?

The discussion continued…

…some voiced suggestions:

- Whatever the copy constructor preserves.
- As long as the two are "equal".
- As long as they're "equivalent".

# What exactly has to be "the Same"?

The discussion continued…

…some voiced suggestions:

- Whatever the copy constructor preserves.
- As long as the two are "equal".
- As long as they're "equivalent".
- "You know what I mean!!"

# What exactly has to be "the Same"?

The discussion continued…

…some voiced suggestions:

- Whatever the copy constructor preserves.
- As long as the two are "equal".
- As long as they're "equivalent".
- "You know what I mean!!"

**Since "purely wording" left solely to the editor!**

2. Understanding Value Semantics

# Not just an “Editorial Issue”?

# Not just an "Editorial Issue"?

What it means for two objects to be "the same" is an important, pervasive, and recurring concept in practical software design.

# Not just an "Editorial Issue"?

What it means for two objects to be "the same" is an important, pervasive, and recurring concept in practical software design.

Based on the notion of "***value***".

# What do we mean by *value*?

# What does a *Copy Constructor* do?

# What does a *Copy Constructor* do?

After copy construction, the resulting object is…

# What does a *Copy Constructor* do?

After copy construction, the resulting object is…

*substitutable* for the original one with respect to “some criteria”.

# What does a *Copy Constructor* do?

After copy construction, the resulting object is…

*substitutable* for the original one with respect to "some criteria".

**What Criteria?**

# Same *Object?*

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);          // ??
```

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);              // ??

assert(0 == b.size());
a.push_back(5.0);

assert(1 == b.size());         // ??
```

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);            // ??

assert(0 == b.size());
a.push_back(5.0);

assert(1 == b.size());       // ??
```

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);              // ??

assert(0 == b.size());
a.push_back(5.0);

assert(1 == b.size());     // ??
```

Java?

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);           // ??

assert(0 == b.size());
a.push_back(5.0);

assert(1 == b.size());      // ??
```

# Same *Object?*

```
std::vector<double> a, b(a);

assert(&a == &b);          // ??

assert(0 == b.size());
a.push_back(5.0);

assert(1 == b.size());     // ??
```

# Same *State?*

# Same *State?*

```
class String {
    char *d_array_p;   // dynamic
    int    d_capacity;
    int    d_length;
  public:
    String();
    String(const String& original);
    // ...
};
```

# Same *State?*

```
class String {
    char *d_array_p;   // dynamic
    int   d_capacit
    int   d_length;
  public:
    String();
    String(const String& original);
    // ...
};
```

What happens if this address is copied?

# Same *Behavior?*

# Same *Behavior?*

If we apply *the same* sequence of operations to both objects, the observable behavior will be *the same*:

# Same *Behavior?*

If we apply *the same* sequence of operations to both objects, the observable behavior will be *the same*:

```
void f(bool x)
{
   std::vector<int> a;
   a.reserve(65536);            // is capacity copied?
   std::vector<int> b(a);          assert(a == b)
```

# Same *Behavior?*

If we apply *the same* sequence of operations to both objects, the observable behavior will be *the same*:

```
void f(bool x)
{
   std::vector<int> a;
   a.reserve(65536);          // is capacity copied?
   std::vector<int> b(a);        assert(a == b)

   a.reserve(65536);          // no reallocation!
   b.reserve(65536);          // memory allocation?
```

# Same *Behavior?*

If we apply *the same* sequence of operations to both objects, the observable behavior will be *the same*:

```
void f(bool x)
{
   std::vector<int> a;
   a.reserve(65536);            // is capacity copied?
   std::vector<int> b(a);          assert(a == b)

   a.reserve(65536);            // no reallocation!
   b.reserve(65536);            // memory allocation?

   a.push_back(5);  b.push_back(5);  // so not empty

   std::vector<int>& r = x ? a : b;
   if (&r[0] == &a[0]) { std::cout << "Hello"; }
   else                { std::cout << "Goodbye"; }
}
```

# Same *What?*

# Same *What?*

What should be “the same” after copy construction?

# Same *What?*

What should be "the same" after copy construction?

(It better be easy to understand.)

# Same *What?*

What should be "the same" after copy construction?

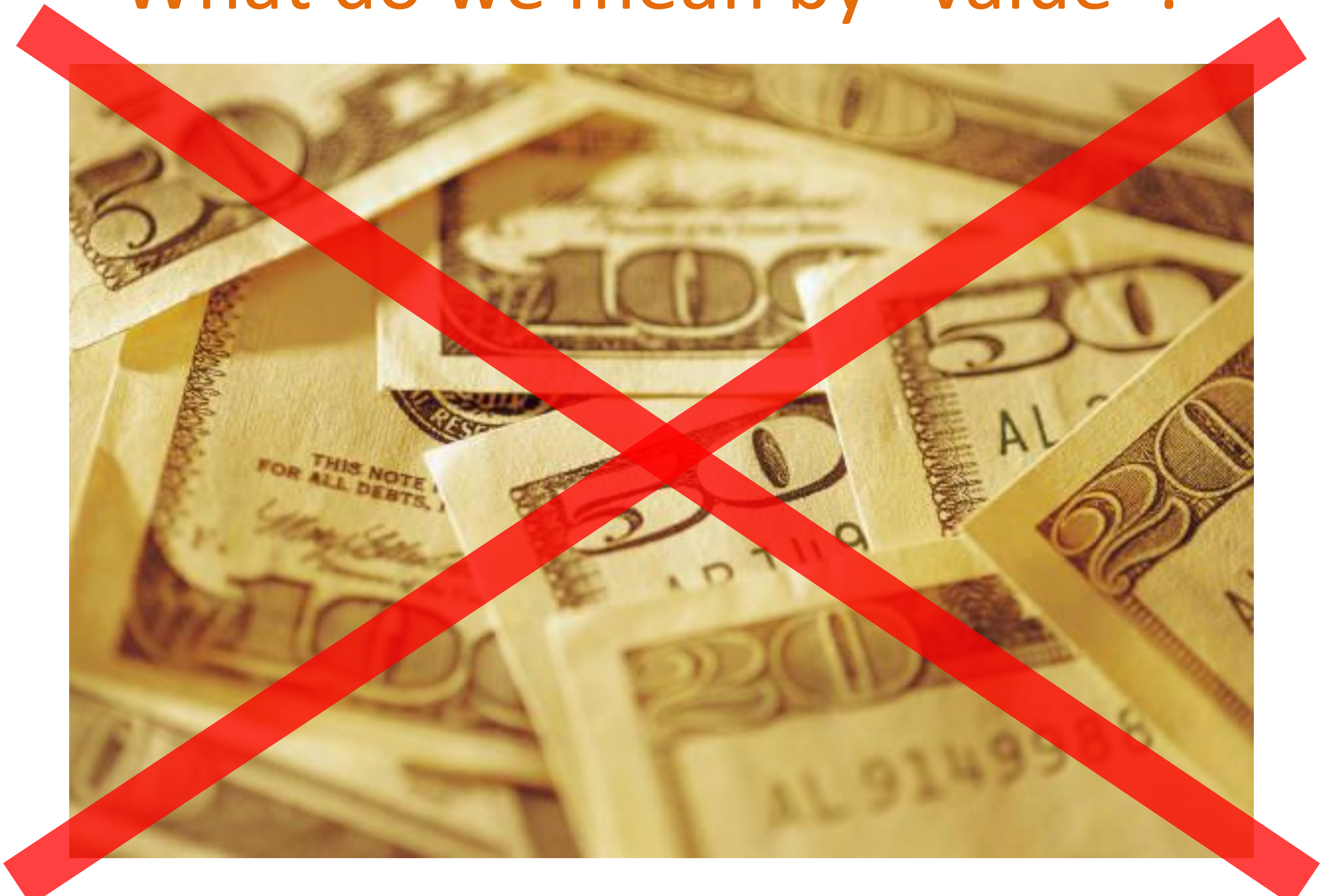
(It better be easy to understand.)

The two objects should *represent* the same ***value!***

# What do we mean by “value”?

## 2. Understanding Value Semantics

# What do we mean by “value”?



2. Understanding Value Semantics

# Mathematical Types

# Mathematical Types

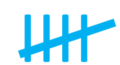
A mathematical type consists of

- A set of globally **unique** *values*
  - Each one describable independently of any particular representation.

# Mathematical Types

A mathematical type consists of

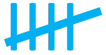
- A set of globally **unique** *values*
  - Each one describable independently of any particular representation.
  - For example, the decimal integer 5:

**5**, ***5***, V, `101(binary)`, **five** , 

# Mathematical Types

A mathematical type consists of

- A set of globally **unique** *values*
  - Each one describable independently of any particular representation.
  - For example, the decimal integer 5:

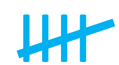
    **5**, ***5***, **V**, `101(binary)`, **five** , 

- A set of *operations* on those values
  - For example: **+**, **-**, **x** **(3 + 2)**

# Mathematical Types

A mathematical type consists of

- A set of globally **unique** *values*
  - Each one describable independently of any particular representation.
  - For example, the decimal integer 5:

    **5**, ***5***, **V**, `101(binary)`, **five** , 

- A set of *operations* on those values
  - For example: **+**, **-**, **x** **(3 + 2)**

Operations will become important shortly!

## 2. Understanding Value Semantics

# C++ Type

# C++ Type

- A C++ type ***may*** represent (an approximation to) an abstract mathematical type:

# C++ Type

- A C++ type ***may*** represent (an approximation to) an abstract mathematical type:
  - For example: The C++ type `int` represents (an approximation to) the mathematical type *integer*.

# C++ Type

- A C++ type ***may*** represent (an approximation to) an abstract mathematical type:
  - For example: The C++ type `int` represents (an approximation to) the mathematical type *integer*.
- An object of such a C++ type represents one of (a subset of) the ***globally unique*** values in the set of that abstract *mathematical* type.

# C++ Type

- A C++ type ***may*** represent (an approximation to) an abstract mathematical type:
  - For example: The C++ type `int` represents (an approximation to) the mathematical type *integer*.
- An object of such a C++ type represents one of (a subset of) the ***globally unique*** values in the set of that abstract *mathematical* type.
- The C++ object is **just another representation** of that *globally unique*, abstract ***value***, e.g., 5.

# So, what do we mean by "value"?

```
class Date {
    short d_year;
    char  d_month;
    char  d_day;
  public:
    // …
    int year();
    int month();
    int day();
};
```

## 2. Understanding Value Semantics

# So, what do we mean by “value”?

```
class Date {
     short d_year;
     char  d_month;
     char  d_day;
  public:
      // …
     int year();
     int month();
     int day();
};
```

# So, what do we mean by “value”?

```
class Date {
     short d_year;
     char  d_month;
     char  d_day;
  public:
      // …
     int year();
     int month();
     int day();
};
```

# So, what do we mean by “value”?

```
class Date {
    short d_year;
    char  d_month;
    char  d_day;
  public:
    // …
    int year() const;
    int month() const;
    int day() const;
};
```

# So, what do we mean by “value”?

```
class Date {
    short d_year;
    char  d_month;
    char  d_day;
  public:
    // …
    int year();
    int month();
    int day();
};
```

```
class Date {

    int d_serial;

  public:
    // …
    int year();
    int month();
    int day();
};
```

# So, what do we mean by “value”?

```
class Date {
    short d_year;
    char  d_month;
    char  d_day;
  public:
    // ...
    int year();
    int month();
    int day();
};
```

```
class Date {
    int d_serial;
  public:
    //
    int year();
    int month();
    int day();
};
```

# So, what do we mean by “value”?

## Salient Attributes

```
int year();
int month();
int day();
```

# So, what do we mean by “value”?

## Salient Attributes

The documented set of (observable) named attributes of a type `T` that must respectively “have” (refer to) *the same* value in order for two instances of `T` to “have” (refer to) *the same* value.

# So, what do we mean by “value”?

```
class Time {
     char  d_hour;
     char  d_minute;
     char  d_second;
     short d_millisec;
  public:
    // …
    int hour();
    int minute();
    int second();
    int millisecond();
};
```

```
class Time {
    int d_mSeconds;




  public:
    // …
    int hour();
    int minute();
    int second();
    int millisecond();
};
```

# So, what do we mean by "value"?

```
class Time {
    Internal Representation

  public:
    //
    int hour();
    int minute();
    int second();
    int millisecond();
};
```

**VALUE**

```
class Time {
    Internal Representation

  public:
    //
    int hour();
    int minute();
    int second();
    int millisecond();
};
```

# So, what do we mean by “value”?

## QUESTION:

What would be the simplest overarching mathematical type for which `std::string` and `(const char *)` are both approximations?

# So, what do we mean by "value"?

## QUESTION:

So if they both represent the character sequence "Fred" do they represent the same value?

**This is important!**

# So, what do we mean by “value”?

## QUESTION:

What about integers and integers mod 5?

# So, what do we mean by "value"?

An "interpretation" of a <u>subset</u> of *instance* state.

# So, what do we mean by "value"?

An "interpretation" of a subset of *instance* state.

- The values of the Salient Attributes, and *not* the instance state used to represent them, comprise what we call the ***value*** of an object.

# So, what do we mean by "value"?

An "interpretation" of a subset of *instance* state.

- The values of the Salient Attributes, and *not* the instance state used to represent them, comprise what we call the ***value*** of an object.
- This definition may be ***recursive*** in that a documented Salient Attribute of a type `T` may itself be of type `U` having its own Salient Attributes.

# So, what do we mean by “value”?

```
class Point {
      short int d_x;
      short int d_y;
   public:
     // …
     int x();
     int y();
};
```

# So, what do we mean by "value"?

```
class Point {
    Internal Representation

  public:
    // …
    int x();
    int y();
};
```

# So, what do we mean by "value"?

```
class Point {
```

***Internal Representation***

```
  public:
    // …
    int x();
    int y();
};
```

```
class Box {
    Point d_topLeft;
    Point d_botRight;
  public:
    // …
    Point origin();
    int length();
    int width();

};
```

# So, what do we mean by “value”?

```
class Point {
    Internal Representation

  public:
    // …
    int x();
    int y();
};
```

```
class Box {
    Internal Representation

  public:
    // …
    Point origin();
    int length();
    int width();

};
```

# So, what do we mean by "value"?

```
class Point {
```
***Internal Representation***
```
  public:
    // …
    int x();
    int y();
};
```

```
class Box {
```
***Internal Representation***
```
  public:
    // …
    Point origin();
    int length();
    int width();

};
```

Recursive

# What are “Salient Attributes”?

# What are “Salient Attributes”?

```
class vector {
    T          *d_array_p;
    size_type   d_capacity;
    size_type   d_size;
    // ...
  public:
    vector();
    vector(const vector<T>& orig);
    // ...
};
```

# What are "Salient Attributes"?

Consider `std::vector<int>`:

What are its *salient attributes*?

# What are "Salient Attributes"?

Consider `std::vector<int>`:

What are its *salient attributes*?

1. The number of elements: `size()`.

# What are “Salient Attributes”?

Consider `std::vector<int>`:

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.

# What are "Salient Attributes"?

Consider `std::vector<int>`:

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.
3. What about `capacity()`?

# What are “Salient Attributes”?

Consider `std::vector<int>`:

What are its *salient attributes*?

1. The number of elements: `size()`.
2. The *values* of the respective elements.
3. ~~What about `capacity()`?~~

How is the client supposed to know for sure?

# What are “Salient Attributes”?

**Salient Attributes:**

# What are “Salient Attributes”?

**Salient Attributes:**

1. Are a part of logical design.

# What are "Salient Attributes"?

**Salient Attributes:**

1. Are a part of logical design.
2. *Should* be "natural" & "intuitive".

# What are “Salient Attributes”?

**Salient Attributes:**

1. Are a part of logical design.
2. *Should* be “natural” & “intuitive”
3. <u>Must</u> be documented *explicitly*!

WHERE? HOW?

SOON!

# Why is value important?

# Why are unique values important?

# Why are unique values important?

Input

IPC

Output

Inter-Process Communication

# Why are <u>unique values</u> important?

<u>Abstract ***date*** Type</u>

<u>C++ `Date` Class</u>

# Why are unique values important?

**Abstract *date* Type**

Has an infinite set of valid *date* values.

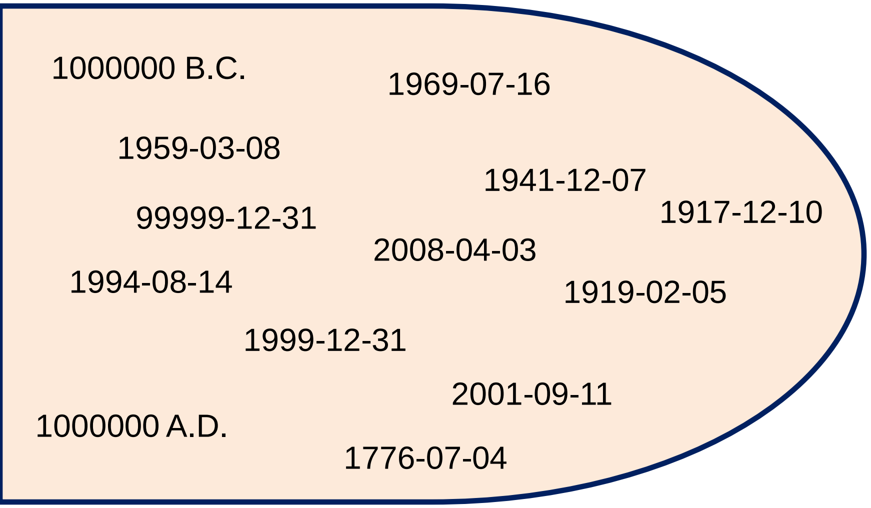
**C++ `Date` Class**

# Why are unique values important?

## Abstract ***date*** Type

## C++ `Date` Class

Has an infinite set of valid *date* values.

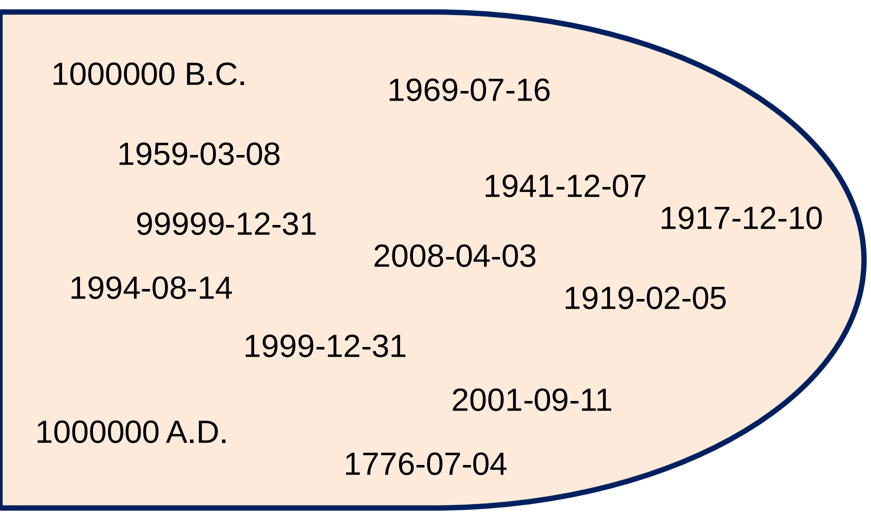



**Globally Unique Values**

# Why are unique values important?

## Abstract ***date*** Type

Has an infinite set of valid *date* values.




**Globally Unique Values**

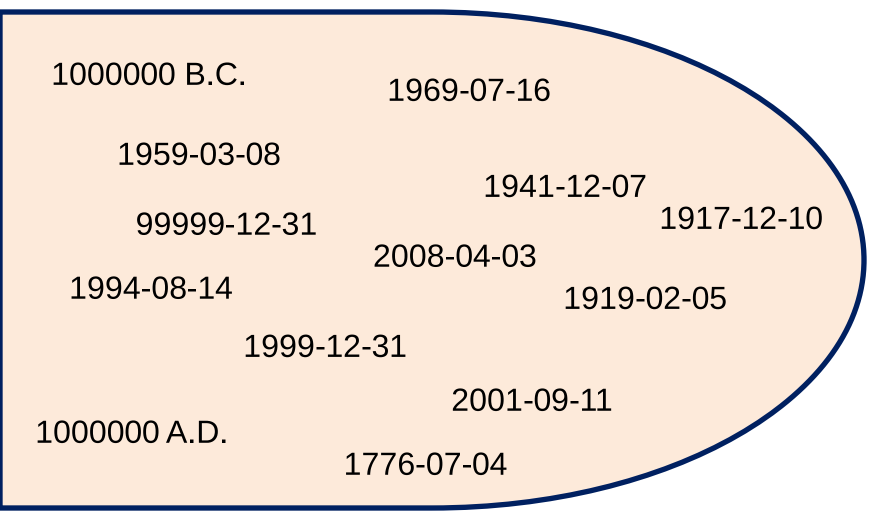
## C++ `Date` Class

Each instance refers to one of (a subset of) these abstract values.

# Why are unique values important?

**Abstract *date* Type**

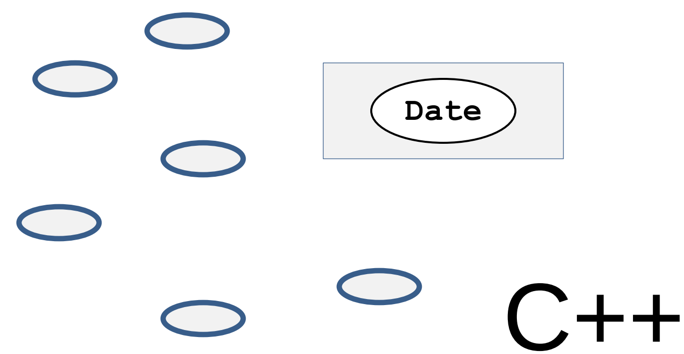
Has an infinite set of valid *date* values.




**Globally Unique Values**

**C++ `Date` Class**

Each instance refers to one of (a subset of) these abstract values.



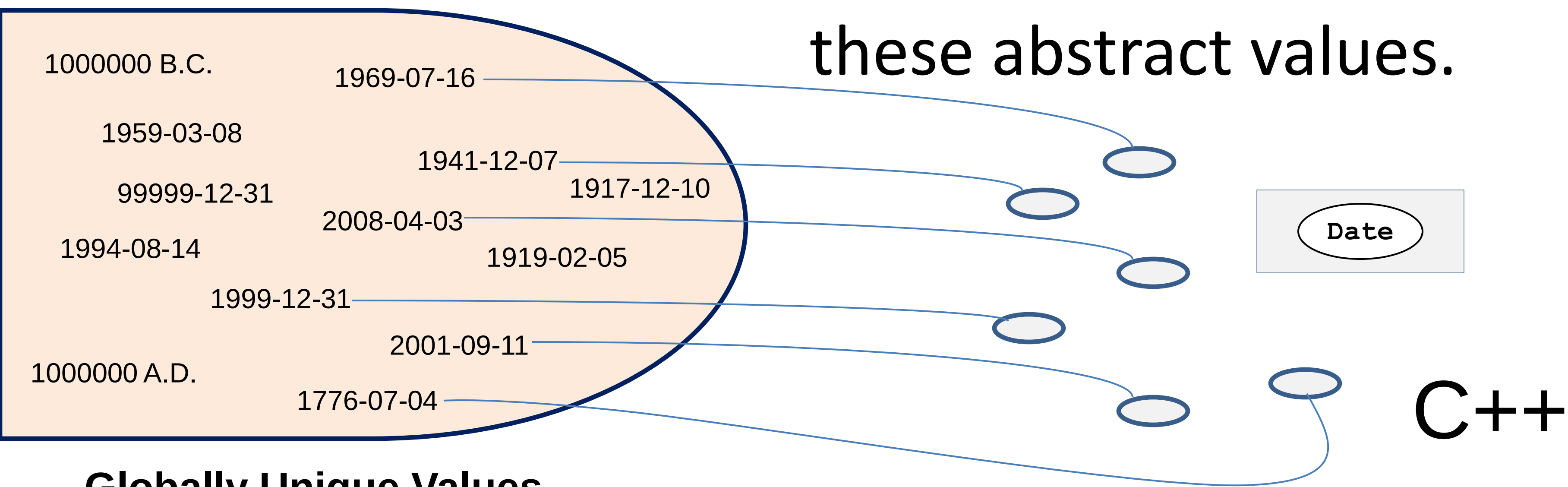
# Why are unique values important?

## Abstract *date* Type

Has an infinite set of valid *date* values.

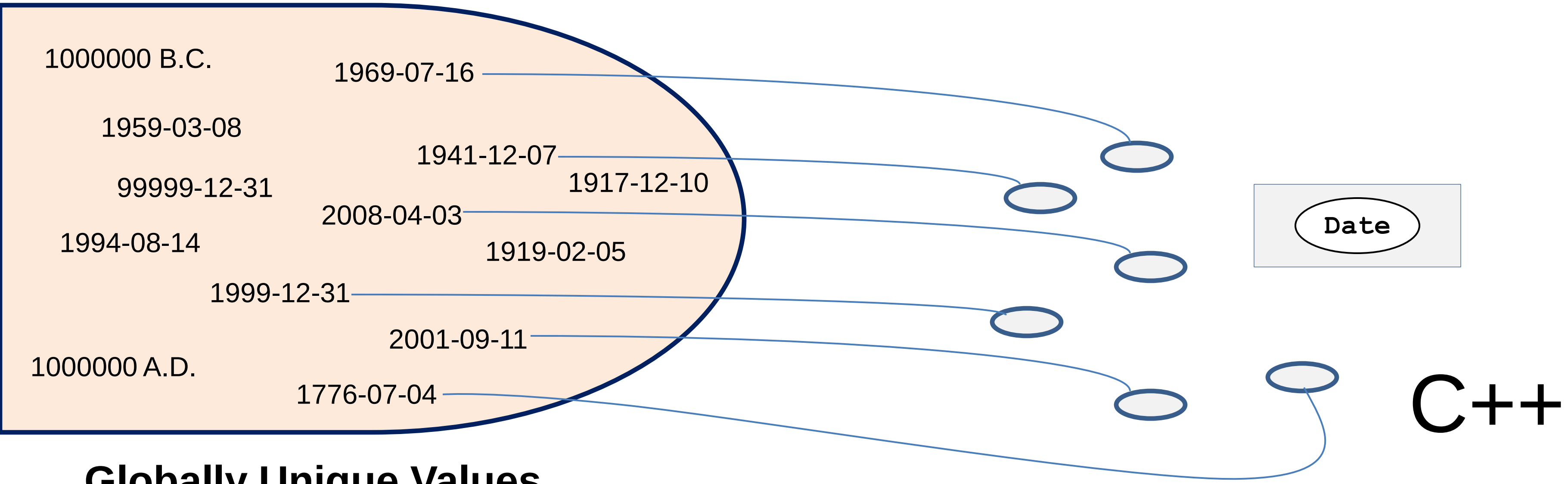
## C++ `Date` Class

Each instance refers to one of (a subset of) these abstract values.



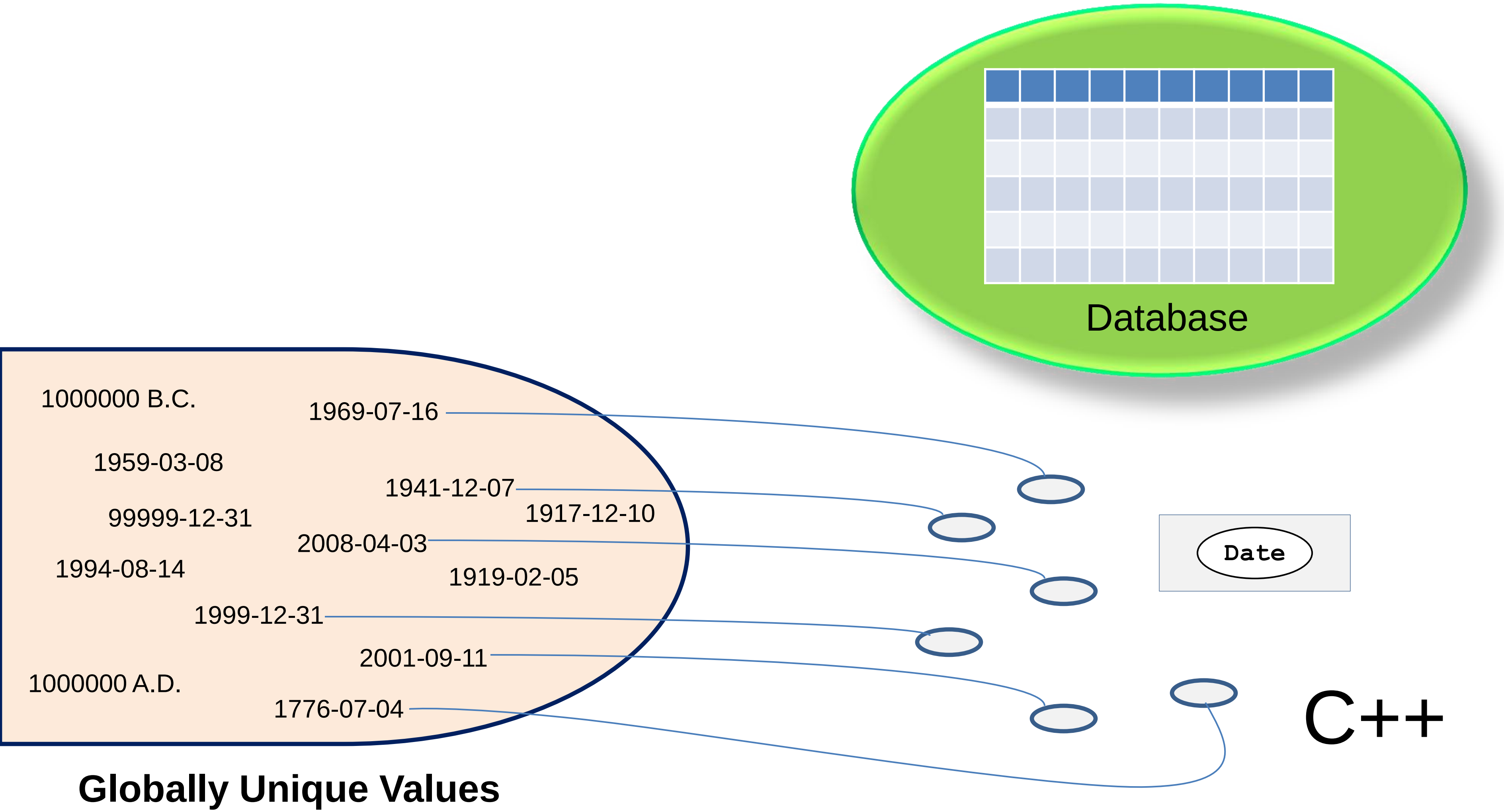

**Globally Unique Values**

# Why are unique values important?

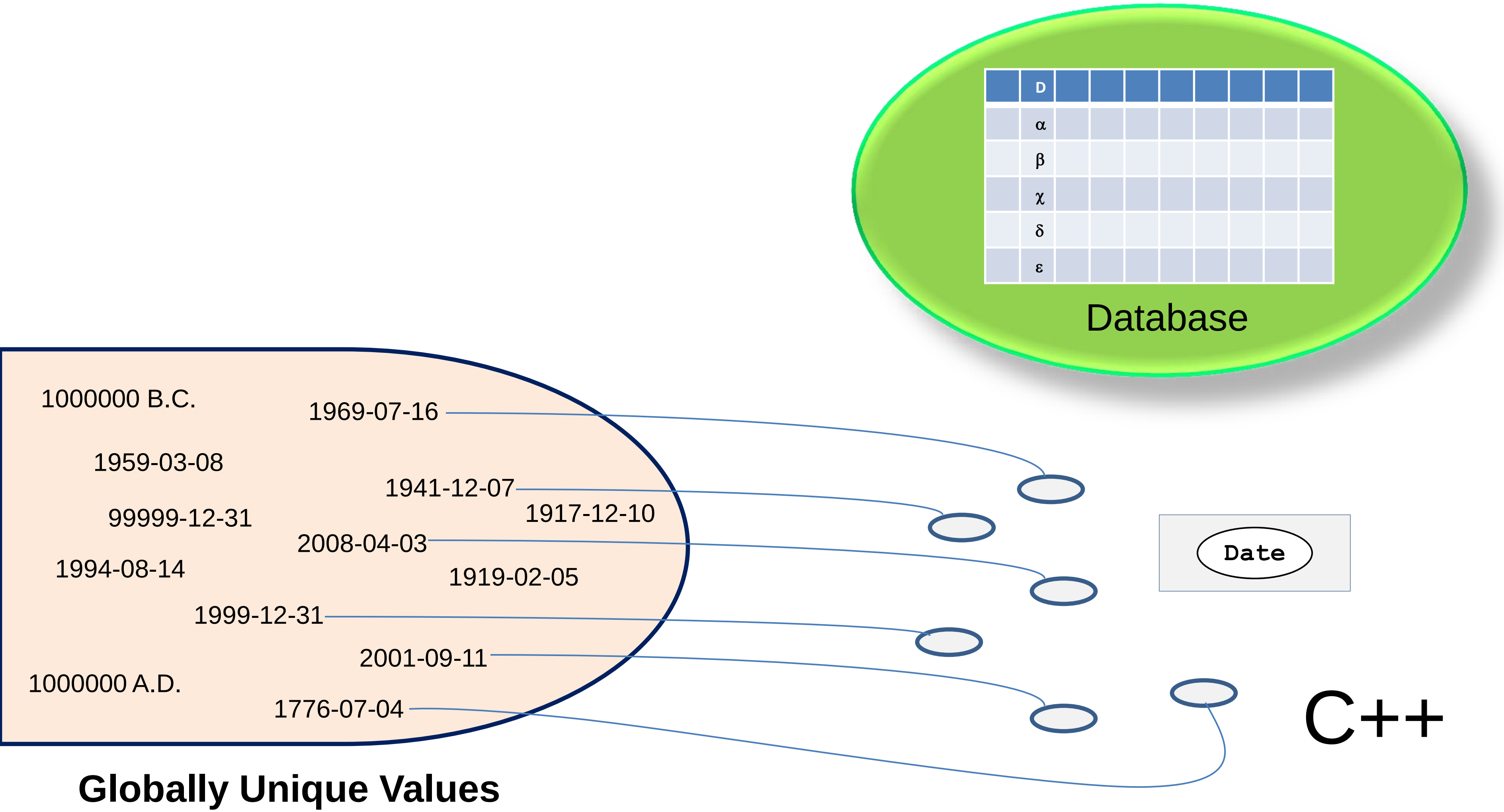



**Globally Unique Values**

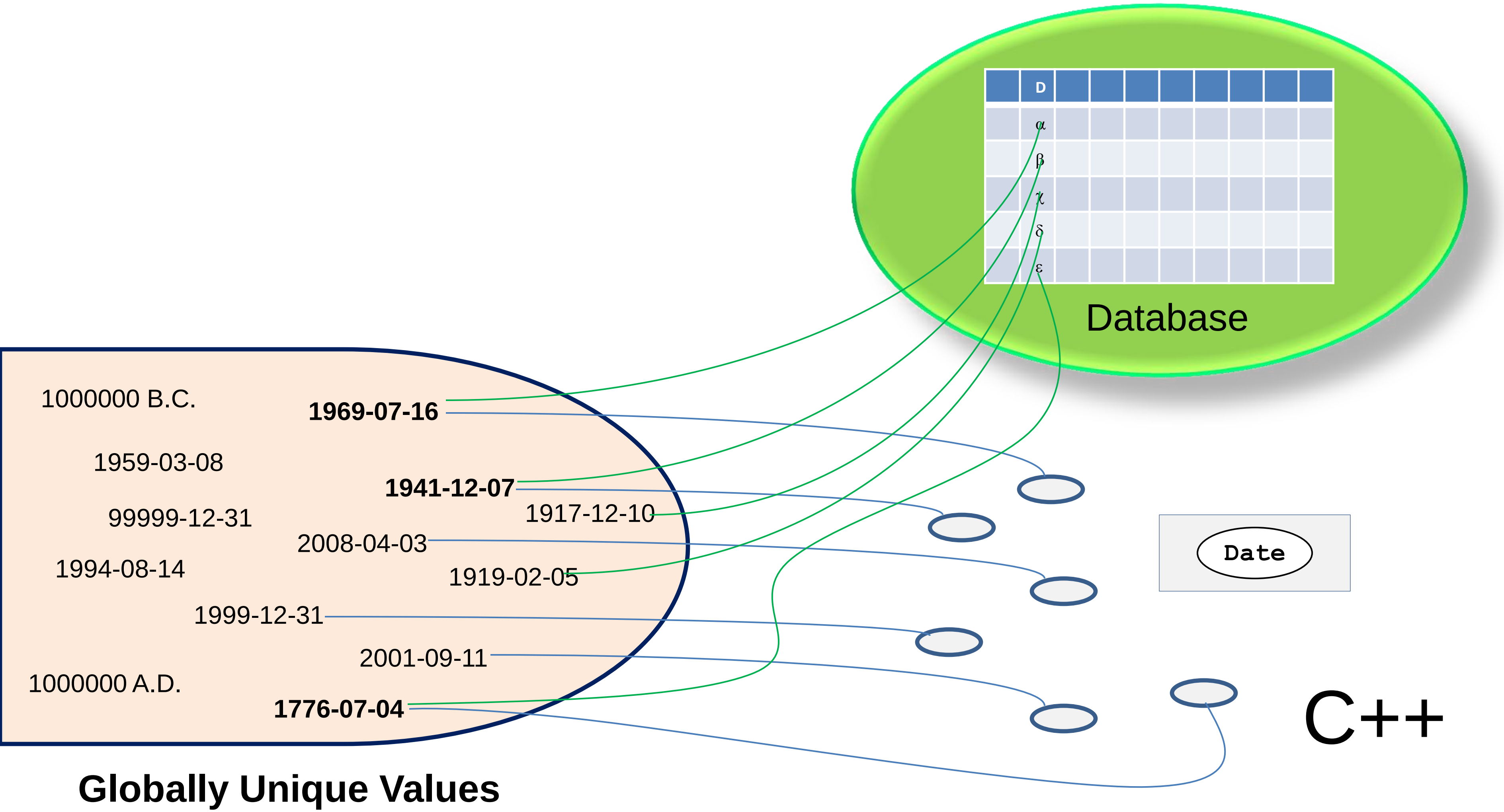
# Why are unique values important?



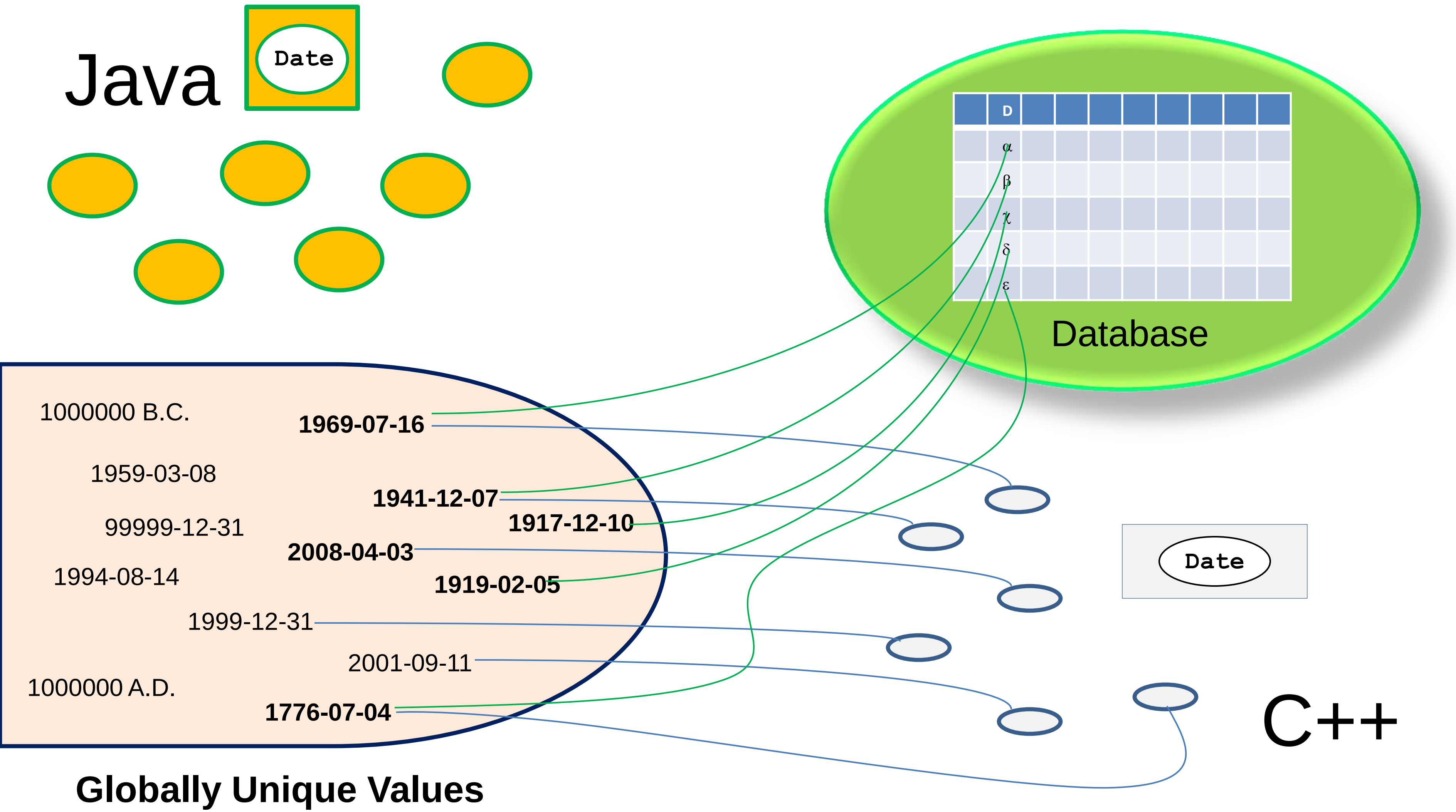
# Why are unique values important?



# Why are unique values important?

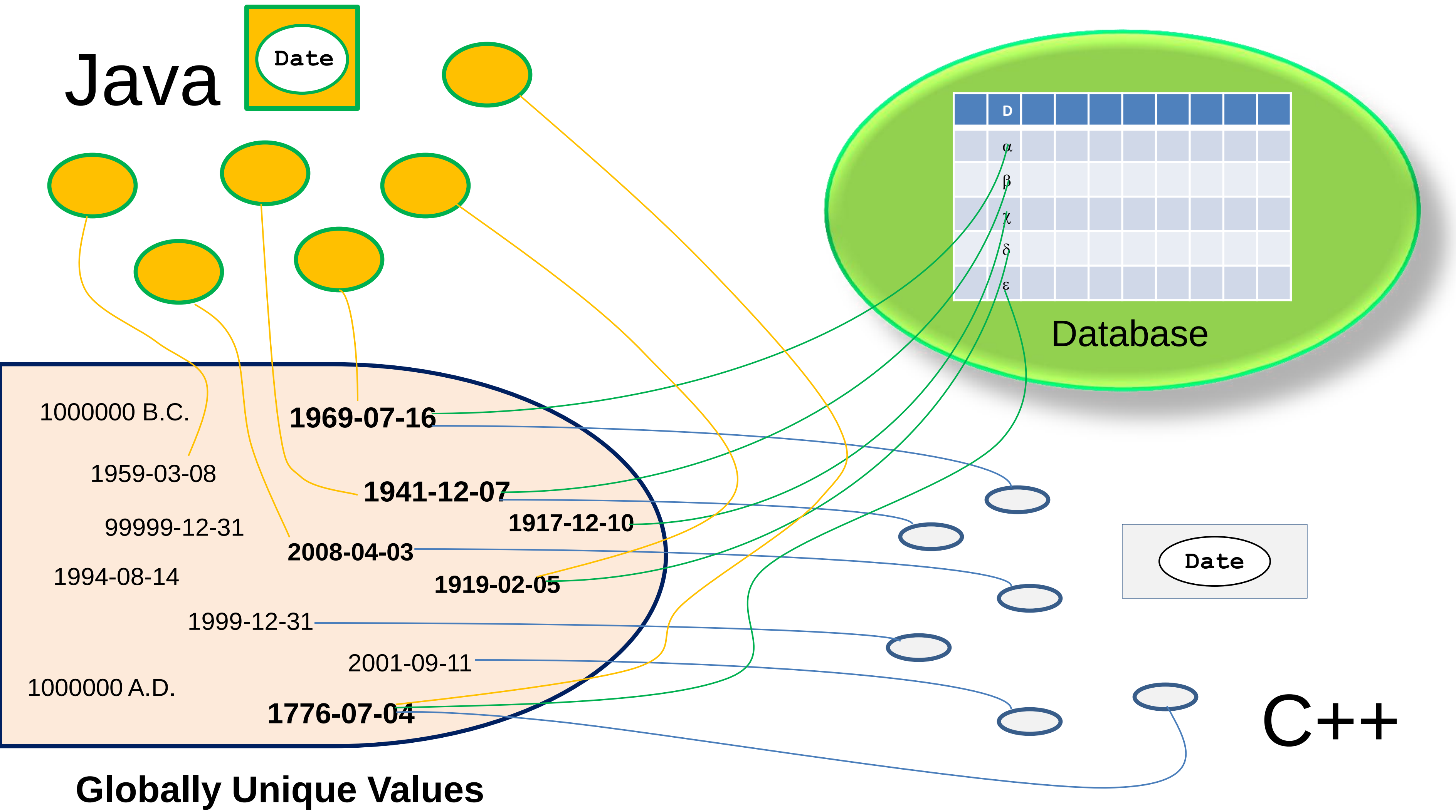


# Why are unique values important?



## 2. Understanding Value Semantics

# Why are unique values important?



# Why are unique values important?

(Not *just* an academic exercise.)

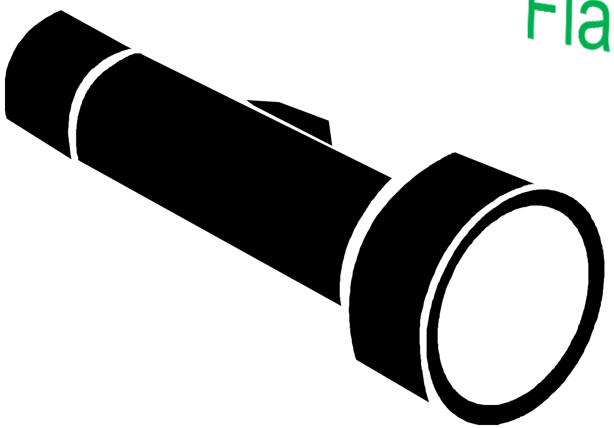
# Why are unique values important?

(Not *just* an academic exercise.)

When we communicate a value outside of a running process, we know that everyone is referring to "***the same***" value.

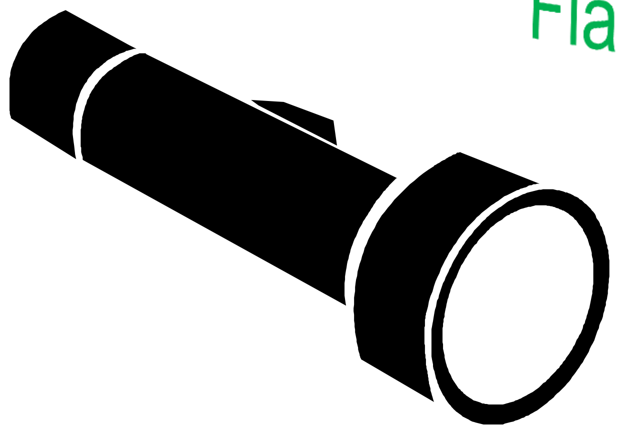
Which types are naturally value types?

# Does **state** *always* imply a "**value**"?

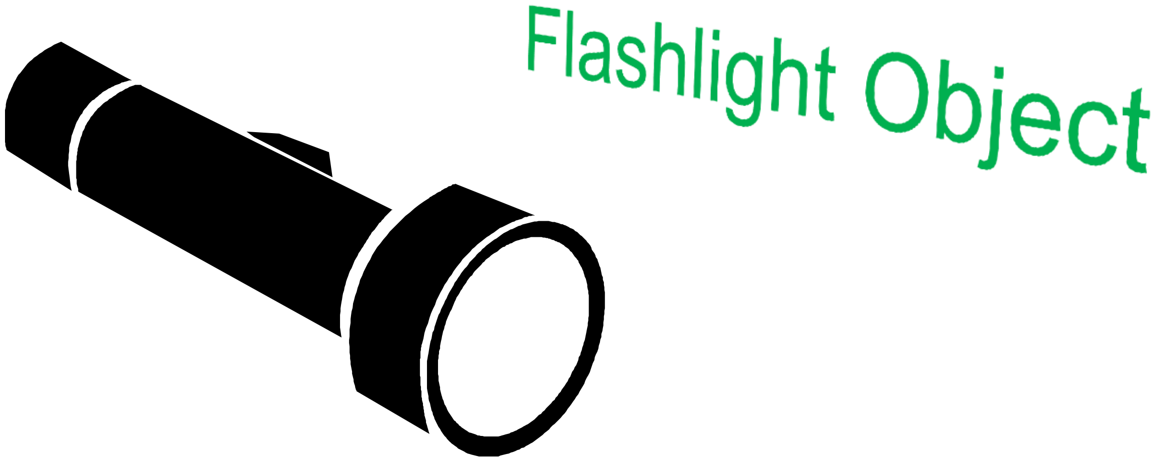


# Does **state** *always* imply a "**value**"?

# Does **state** *always* imply a “**value**”?

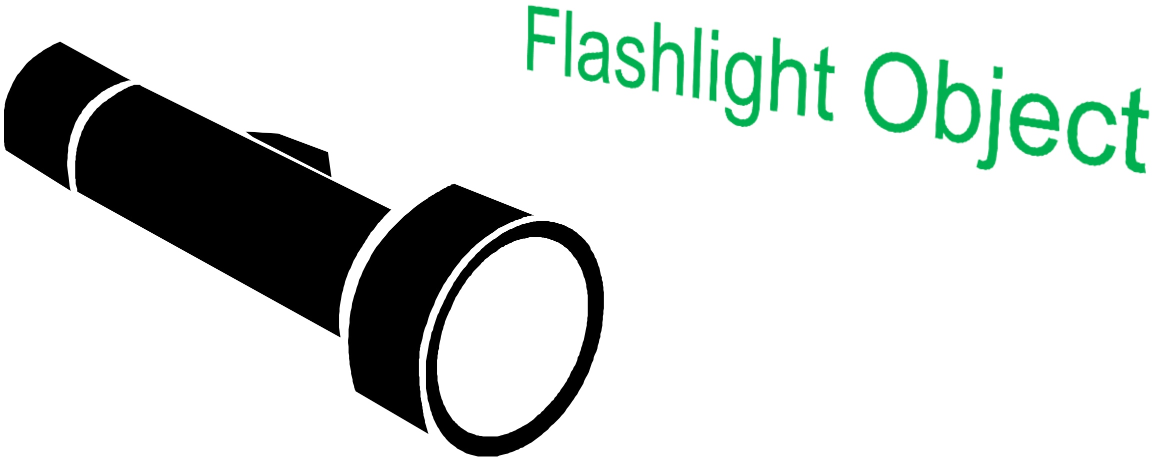


# Does **state** *always* imply a "**value**"?



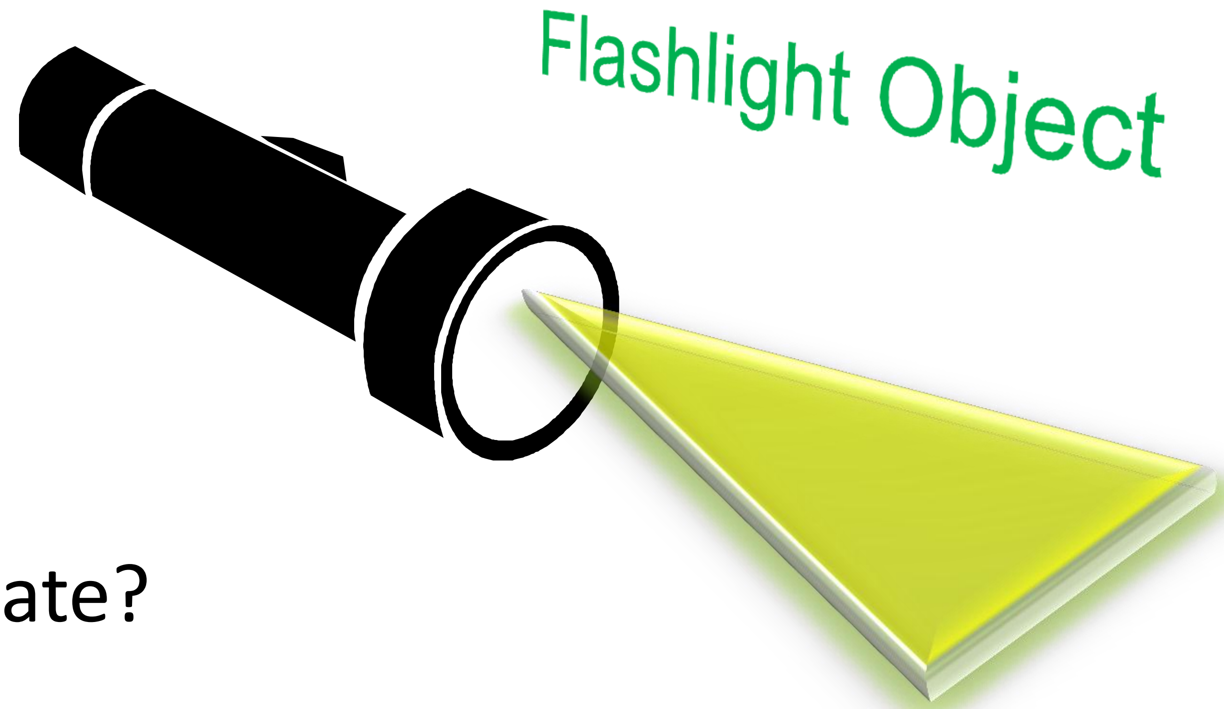

What is its state?

# Does **state** *always* imply a "**value**"?



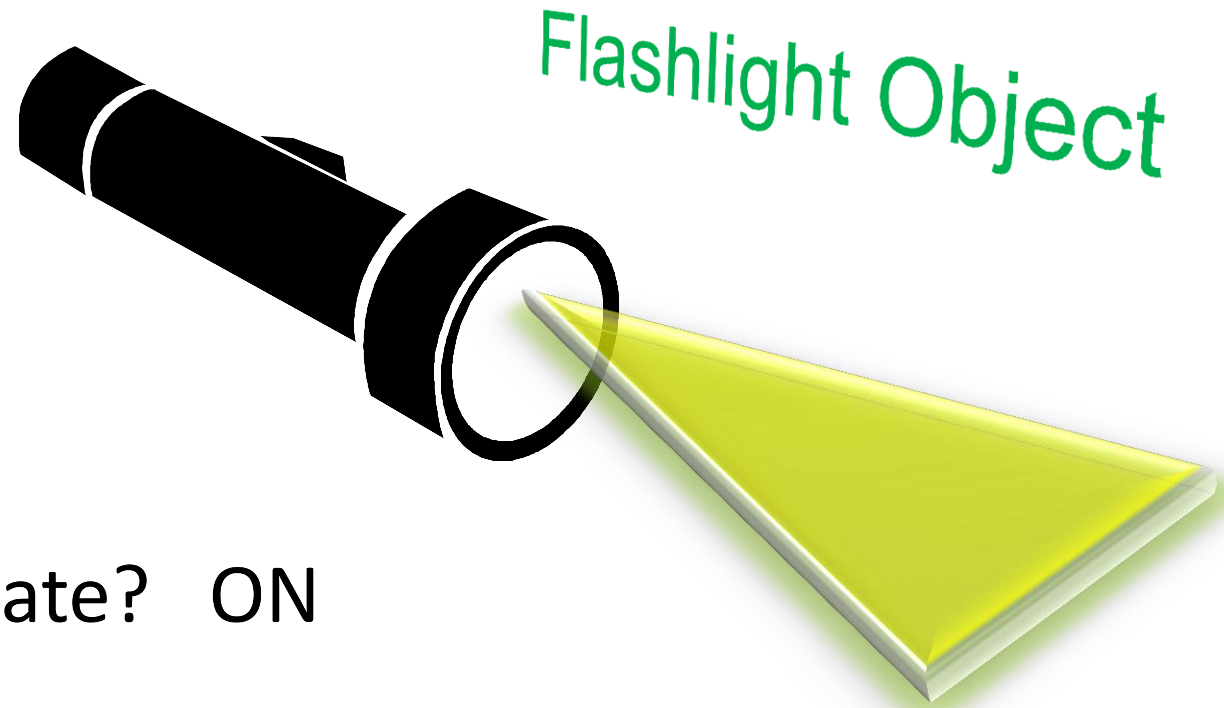

What is its state? OFF

# Does **state** *always* imply a "**value**"?



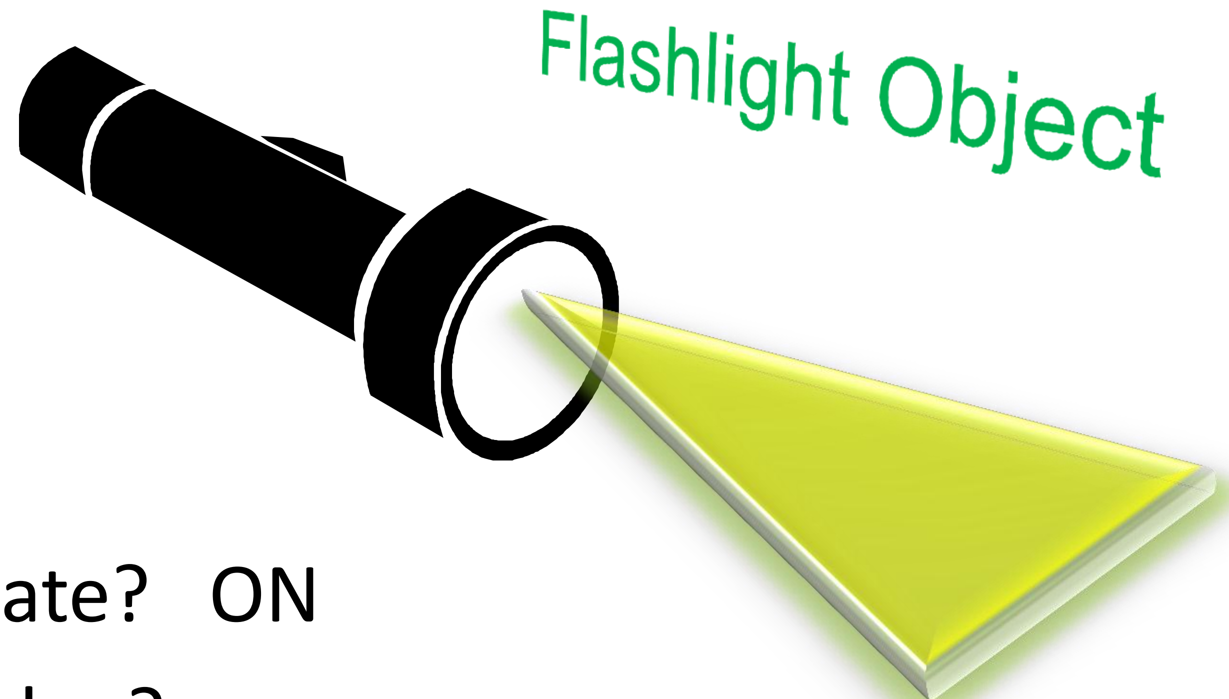

What is its state?

# Does **state** *always* imply a "**value**"?




What is its state? ON

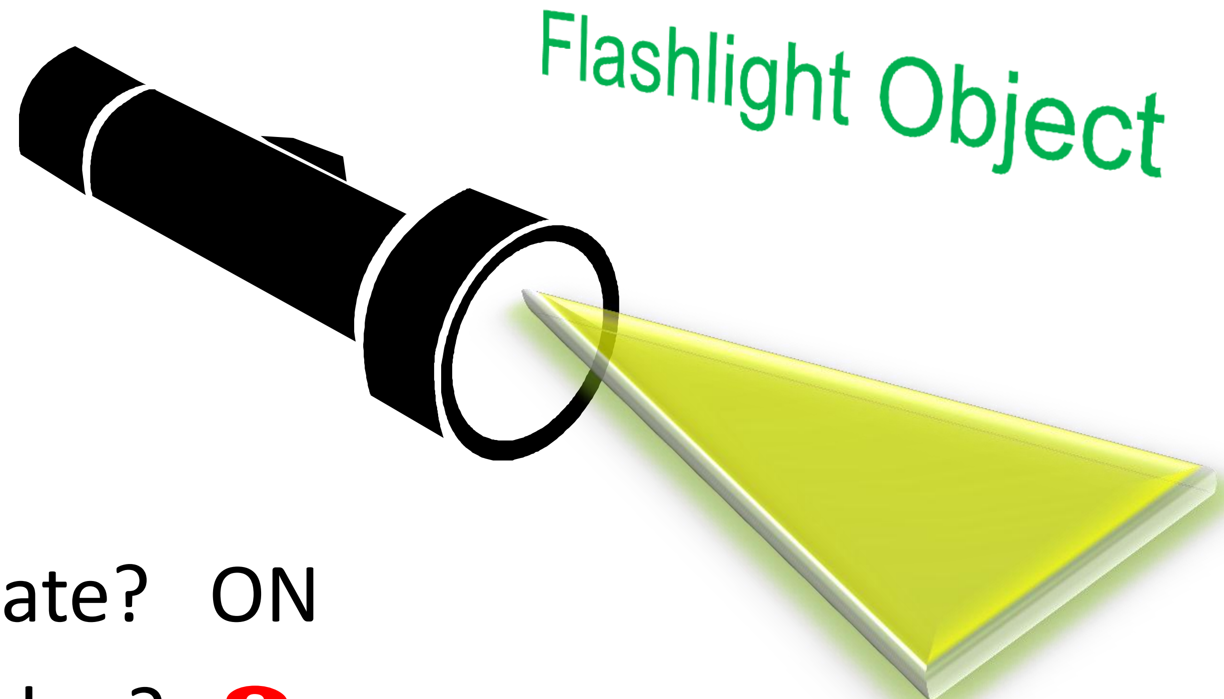
# Does **state** *always* imply a "**value**"?




What is its state? ON

What is its value?

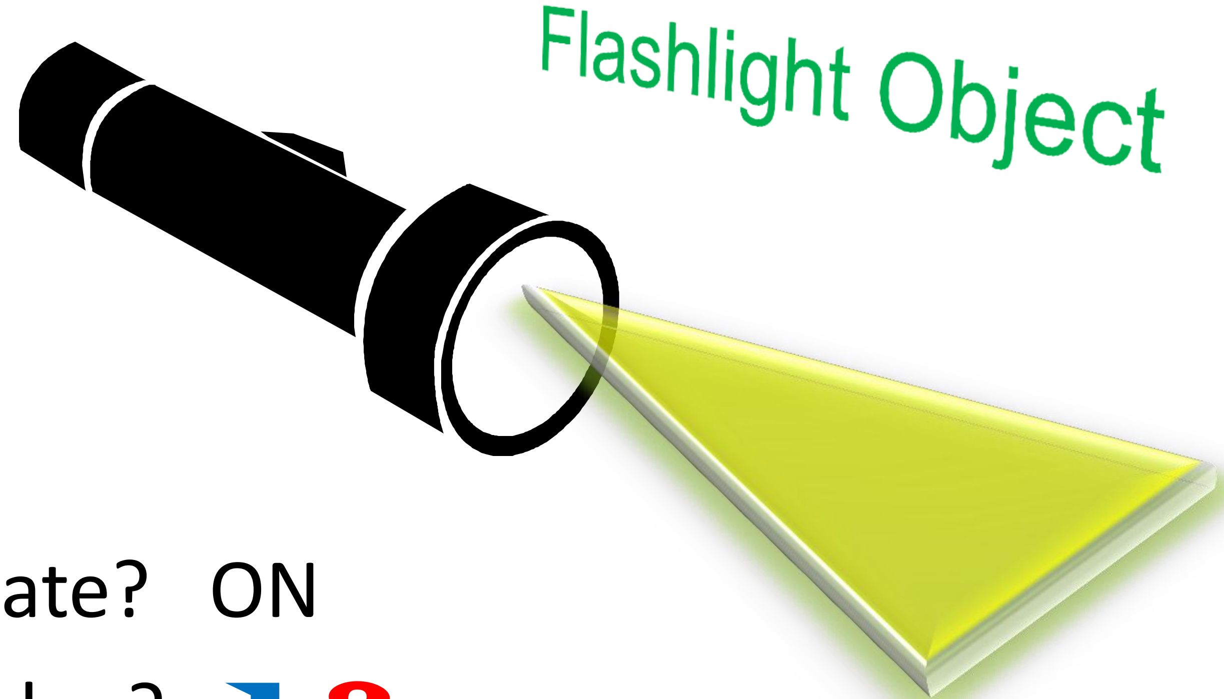
# Does **state** *always* imply a "**value**"?




What is its state? ON

What is its value? ?

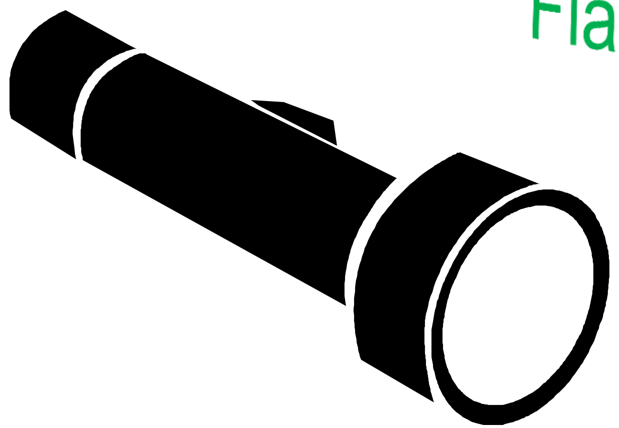
# Does **state** *always* imply a "**value**"?




What is its state? ON

What is its value? ❢ ❓

# Does **state** *always* imply a "**value**"?




What is its state? ON

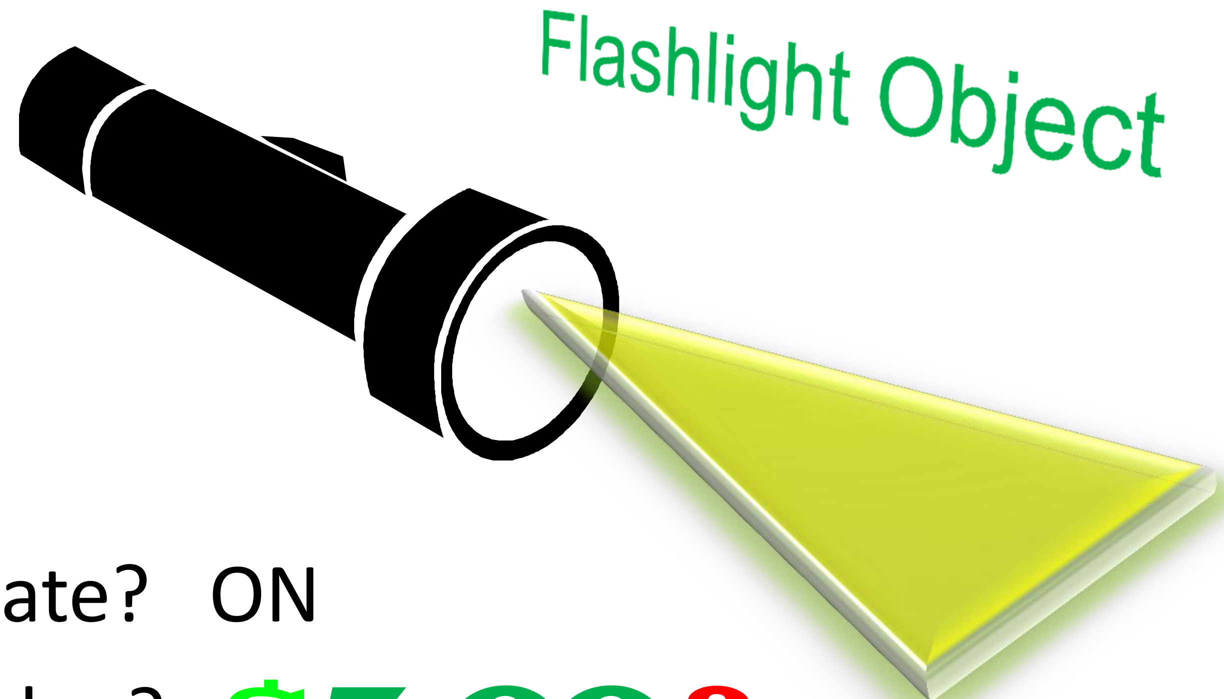
What is its value? **false ?**

# Does **state** *always* imply a "**value**"?


What is its state? ON

What is its value? £5.00 ?

# Does **state** *always* imply a "**value**"?

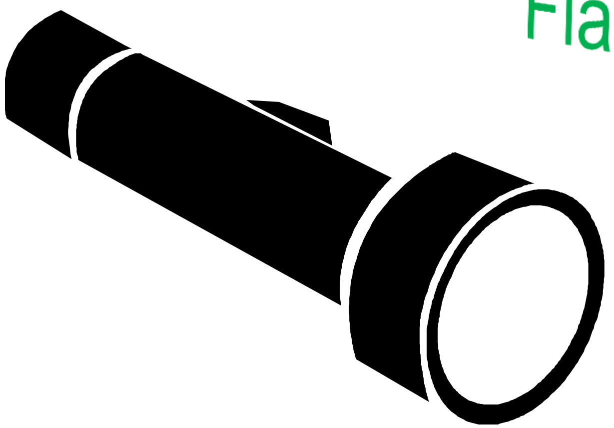



What is its state? ON

What is its value? **$5.00 ?**

Cheap at half the price!

# Does **state** *always* imply a "**value**"?




What is its state? ON

What is its value? ❓

Any notion of "value" here would be artificial!

# Does **state** *always* imply a "**value**"?

Not every ***stateful*** object has an ***obvious*** value.

# Does **state** *always* imply a “**value**”?

Not every ***stateful*** object has an ***obvious*** value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

# Does **state** *always* imply a "**value**"?

<u>Not</u> every ***stateful*** object has an ***obvious*** value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

What would copy construction even *mean* here?

# Does **state** *always* imply a "**value**"?

<u>Not</u> every ***stateful*** object has an ***obvious*** value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

What would copy construction even *mean* here?

We could *invent* some notion of value, but to what end??

# Does **state** *always* imply a "**value**"?

Not every ***stateful*** object has an ***obvious*** value.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex Lock
- Reader/Writer Lock
- Scoped Guard

- Base64 En(De)coder
- Expression Evaluator
- Language Parser
- Event Logger
- Object Persistor
- Widget Factory

# Does **state** *always* imply a "**value**"?

QUESTION:

Suppose we have a thread-safe queue used for inter-task communication: Is it a value type?

# Does **state** *always* imply a "**value**"?

This class is a rare and subtle middle ground.

## QUESTION:

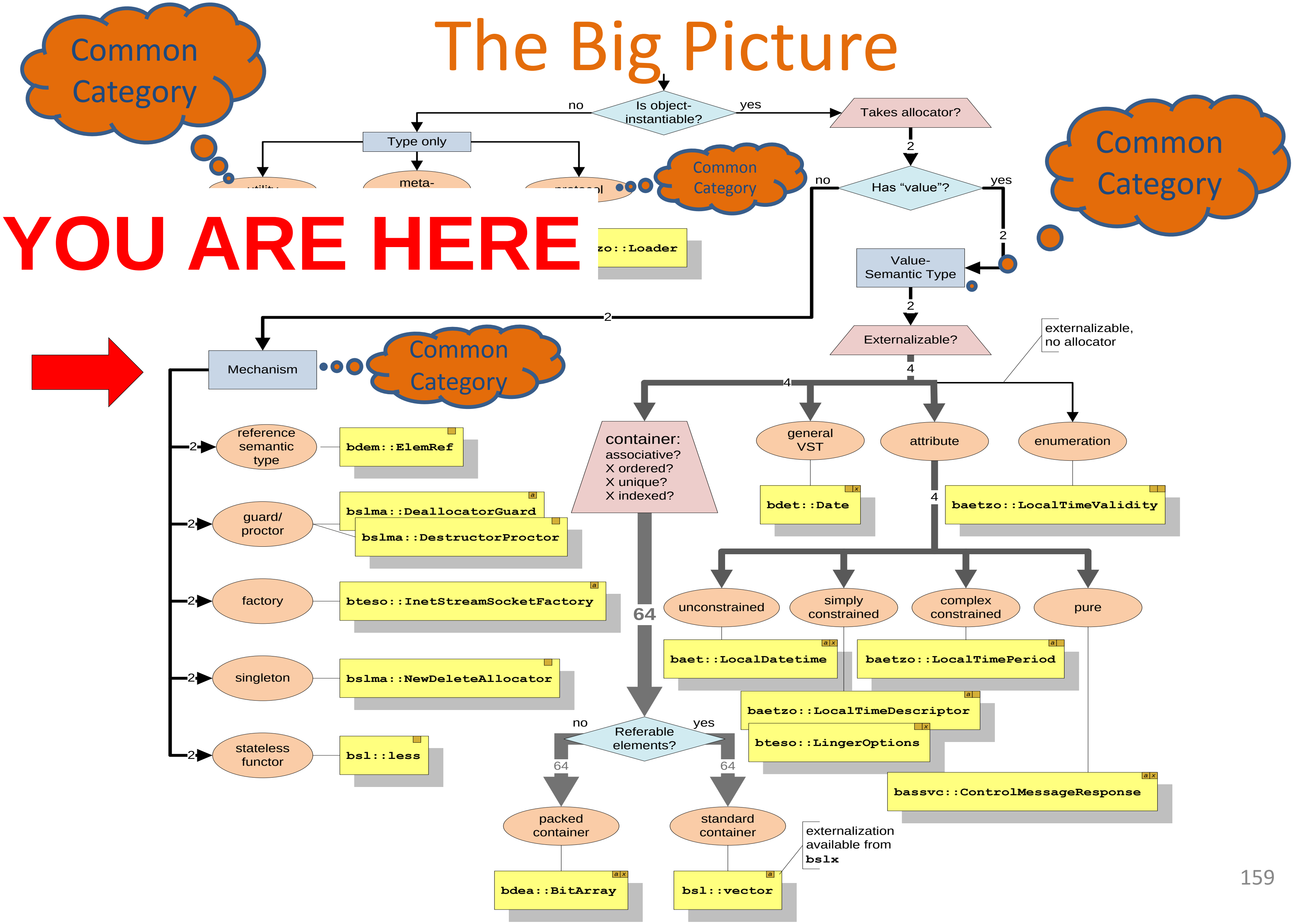
Suppose we have a thread-safe queue used for inter-task communication: Is it a value type? Should this object type support value-semantic syntax?

# Does **state** *always* imply a "**value**"?

**We refer to *stateful* objects that do not represent a value as "Mechanisms".**

## 2. Understanding Value Semantics

# The Big Picture



# Categorizing Object Types

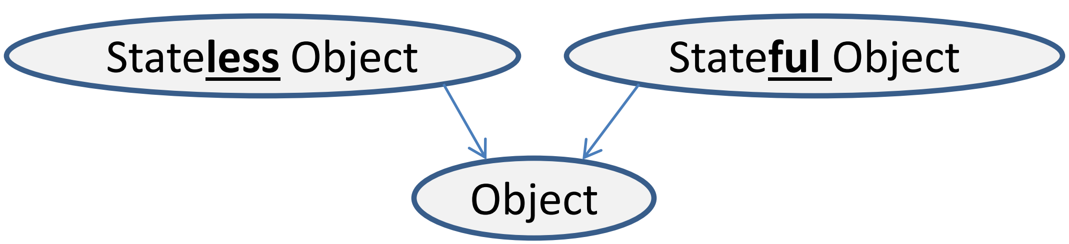
MyObjectType

# Categorizing Object Types

The first question: “Does it have state?”

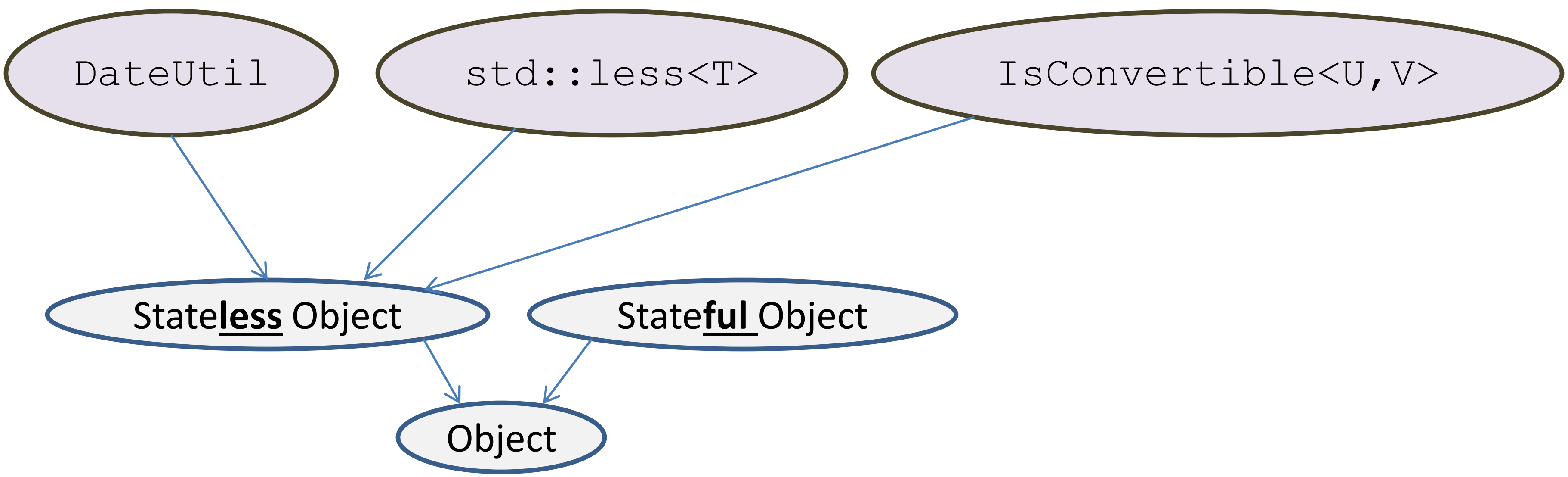
# Categorizing Object Types

The first question: "Does it have state?"



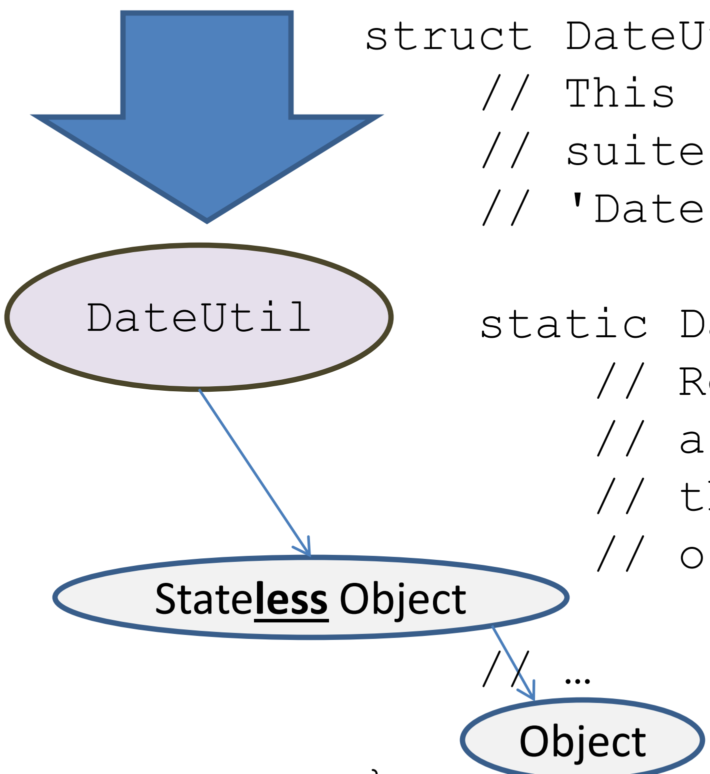
# Categorizing Object Types

The first question: “Does it have state?”



# Categorizing Object Types

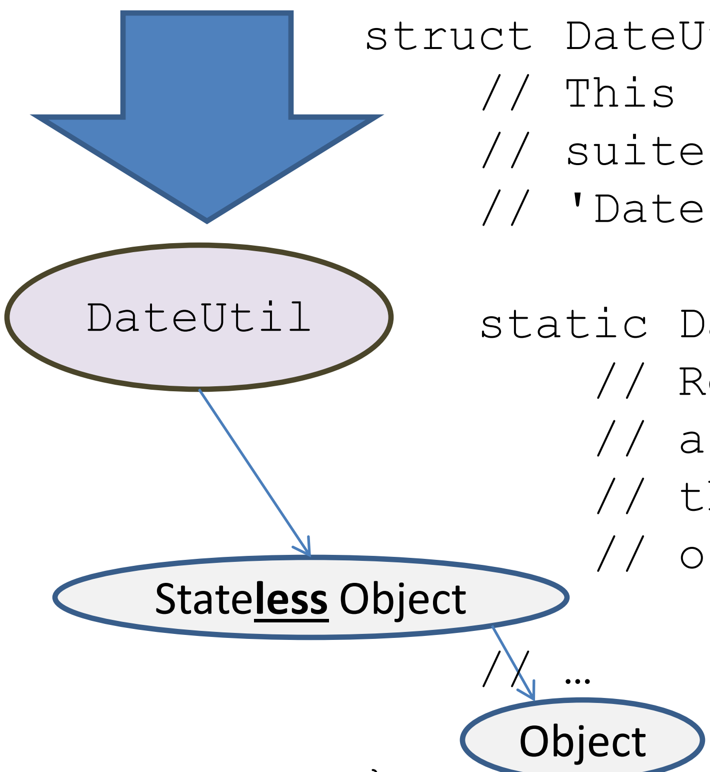
## The first question: "Does it have state?"




```
struct DateUtil {
    // This 'struct' provides a namespace for a
    // suite of pure functions that operate on
    // 'Date' objects.

    static Date lastDateInMonth(const Date& value);
        // Return the last date in the same month
        // as the specified date 'value'.  Note
        // that the particular day of the month
        // of 'value' is ignored.

    // ...
}
```

# Categorizing Object Types

## The first question: “Does it have state?”

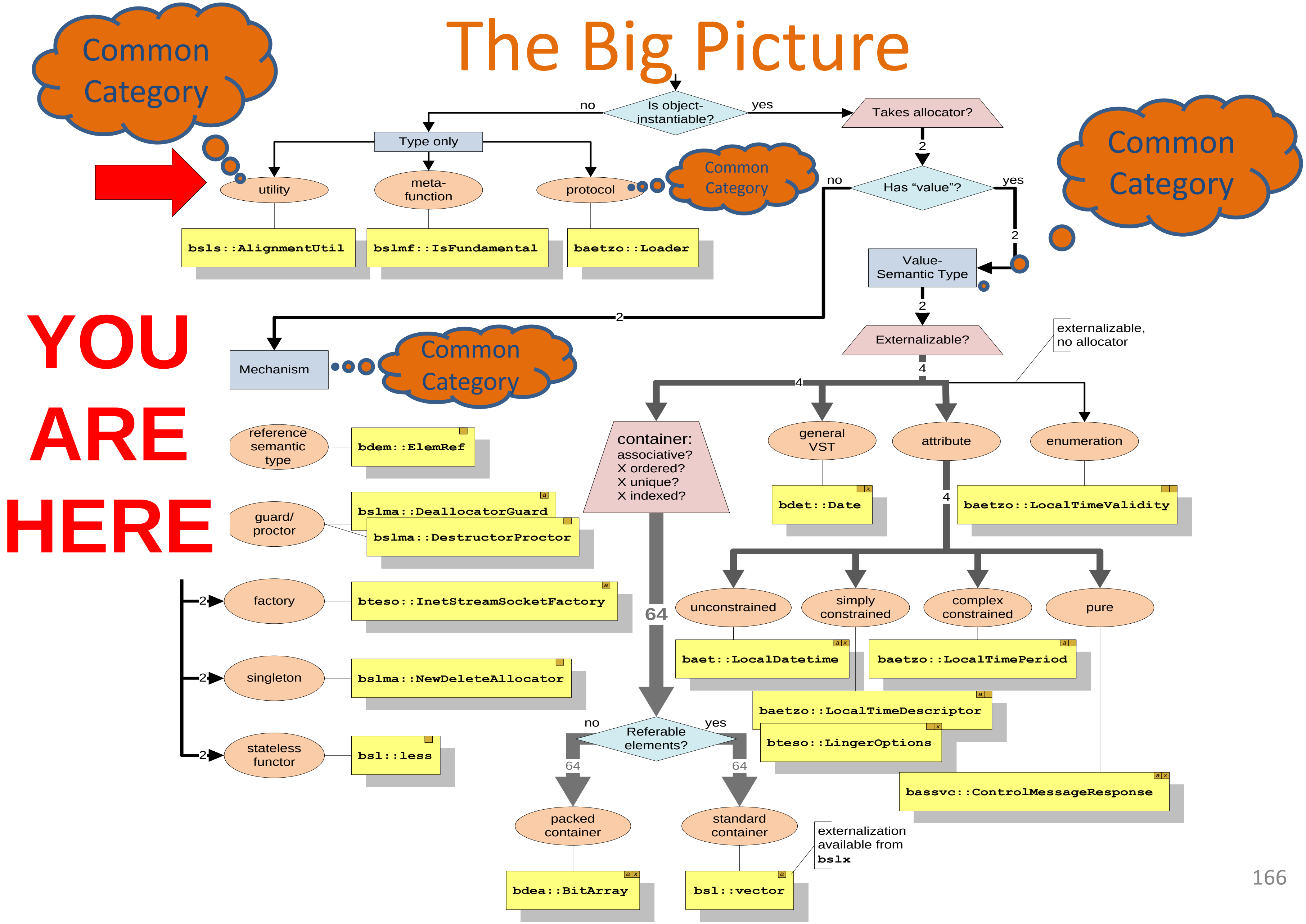



```
struct DateUtil {
    // This 'struct'
    // suite of pure
    // 'Date' object

    static Date last                                        ;
        // Return the
        // as the spe
        // that the p
        // of 'value'

    // ...
}
```

**Utilities are an important class category!**

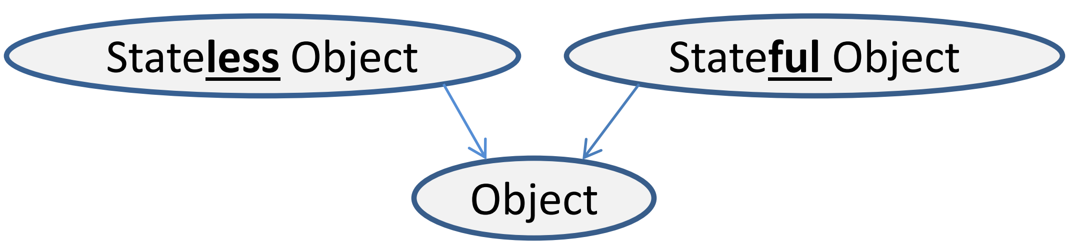
## 2. Understanding Value Semantics

# The Big Picture



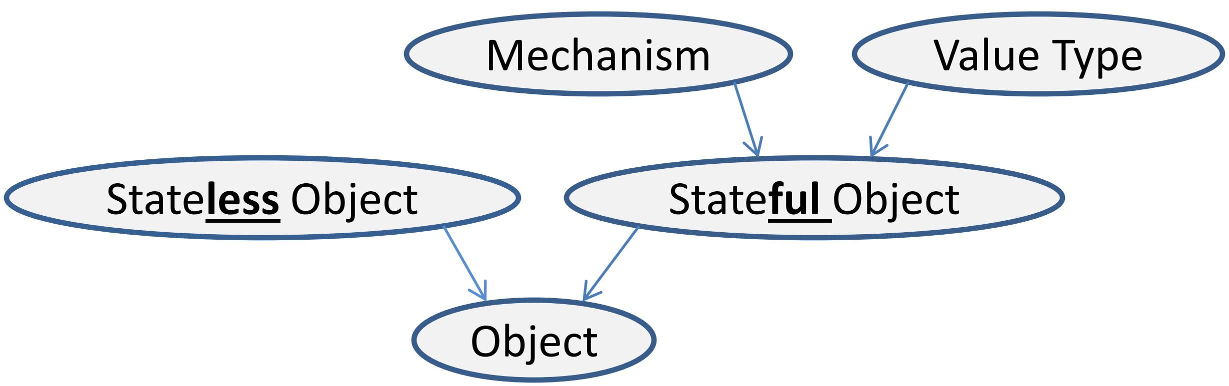
# Categorizing Object Types

The second question: "Does it have value?"



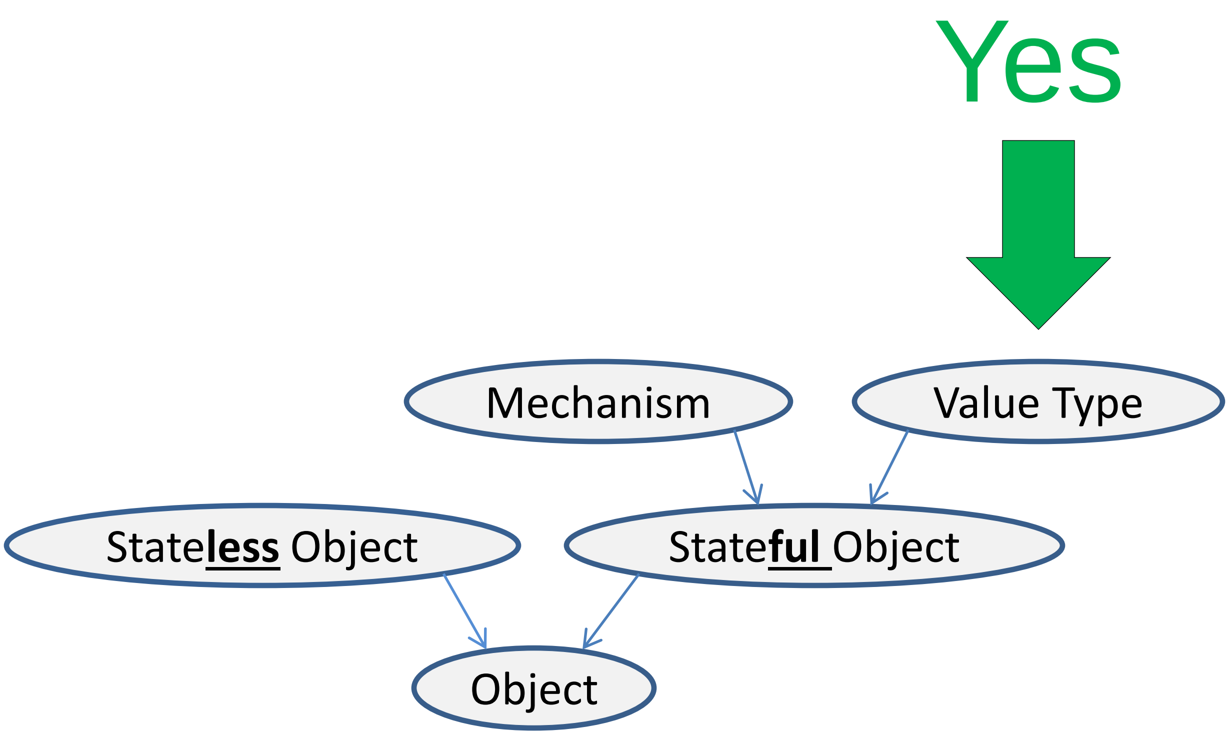
# Categorizing Object Types

The second question: "Does it have value?"



# Categorizing Object Types

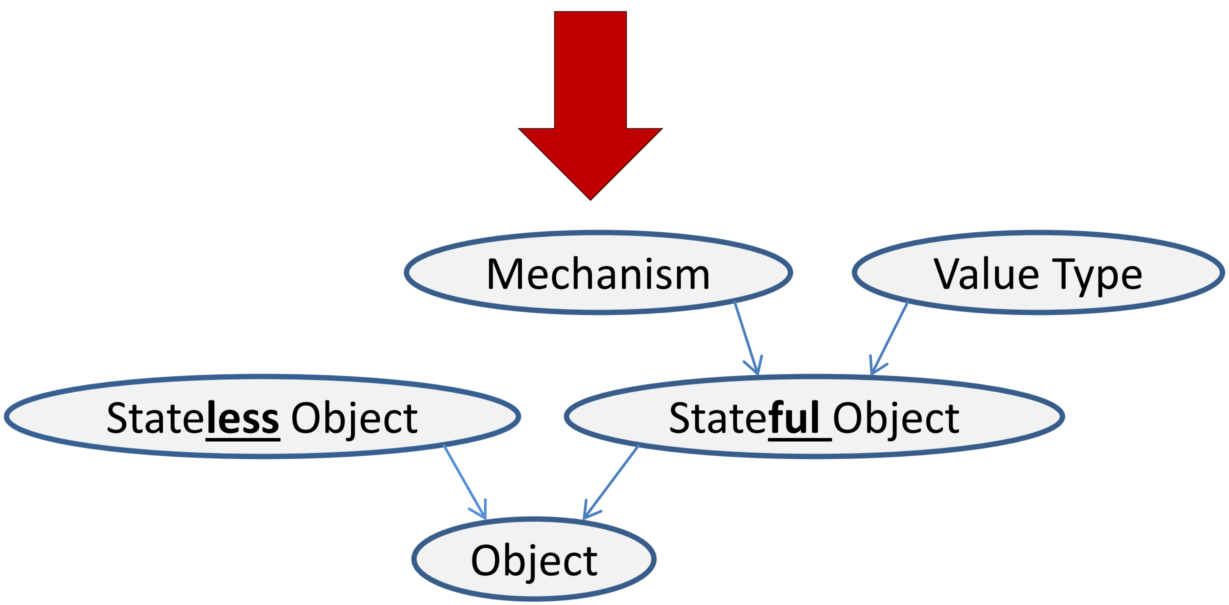
The second question: "Does it have value?"



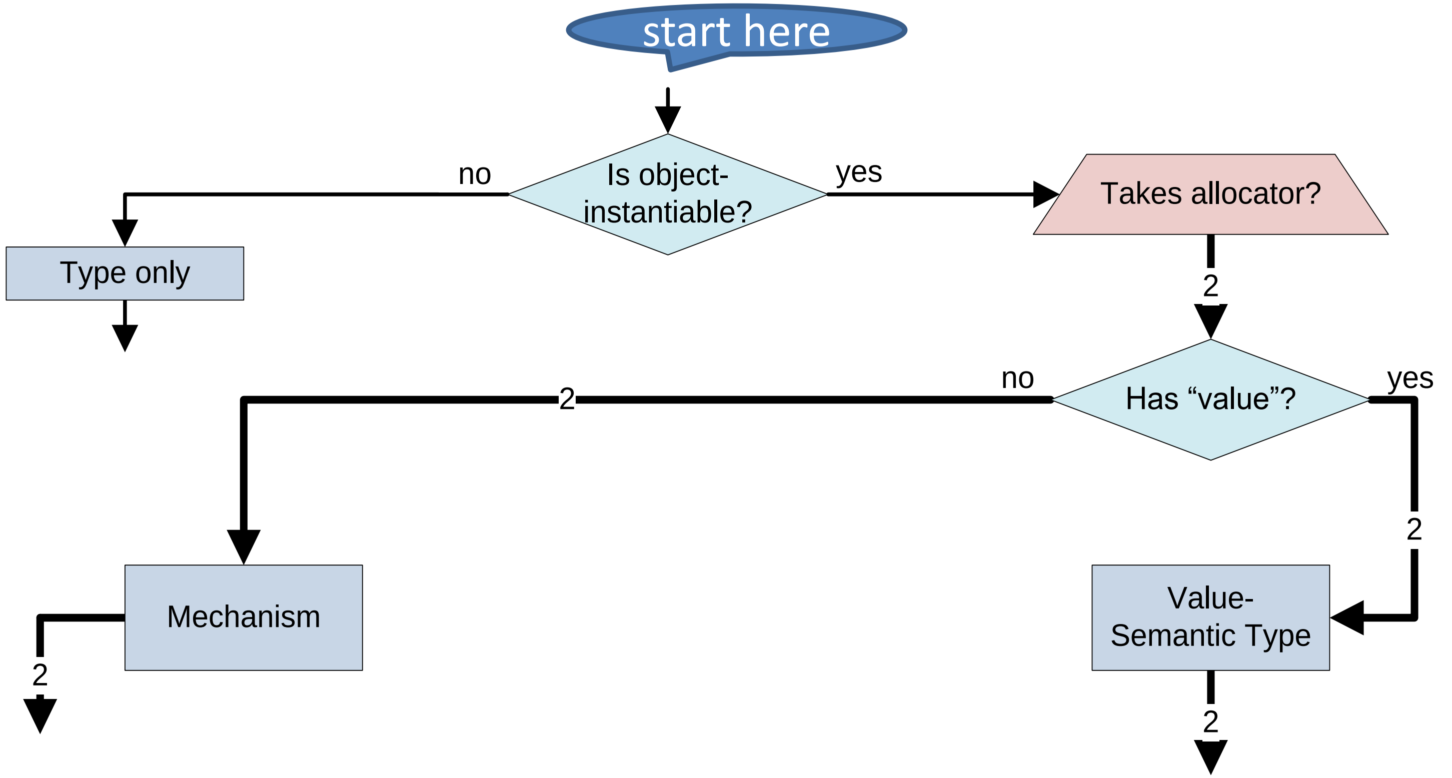
# Categorizing Object Types

The second question: “Does it have value?”

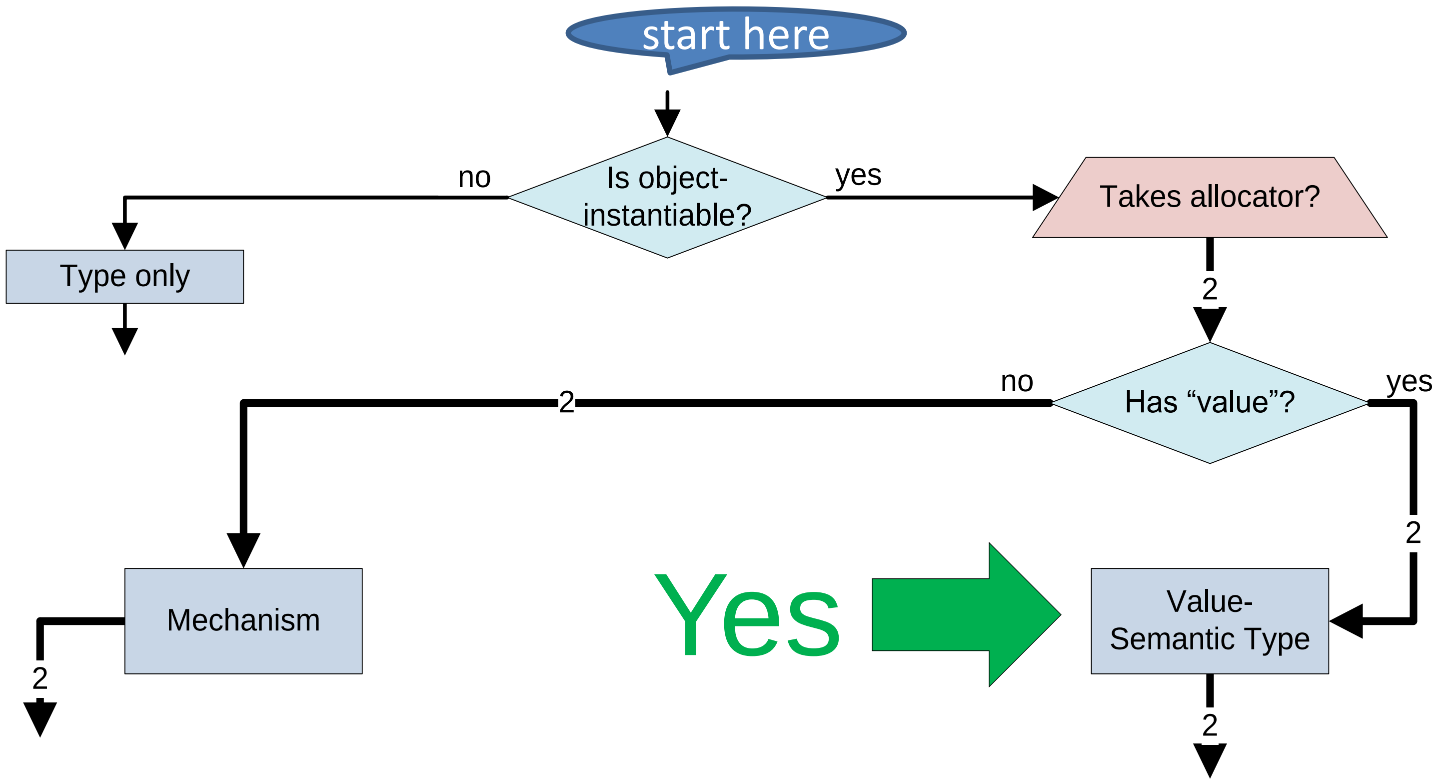
No



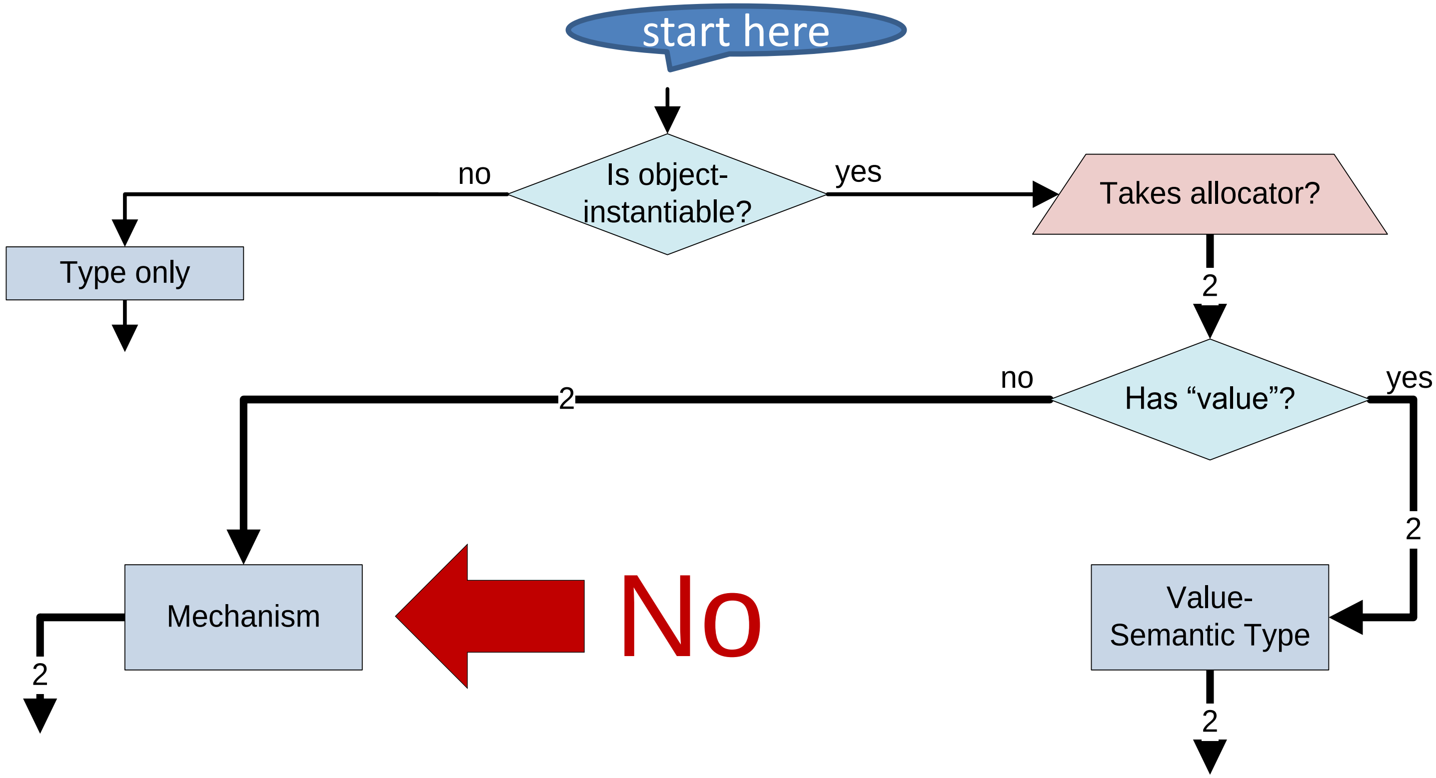
# Top-Level Categorizations



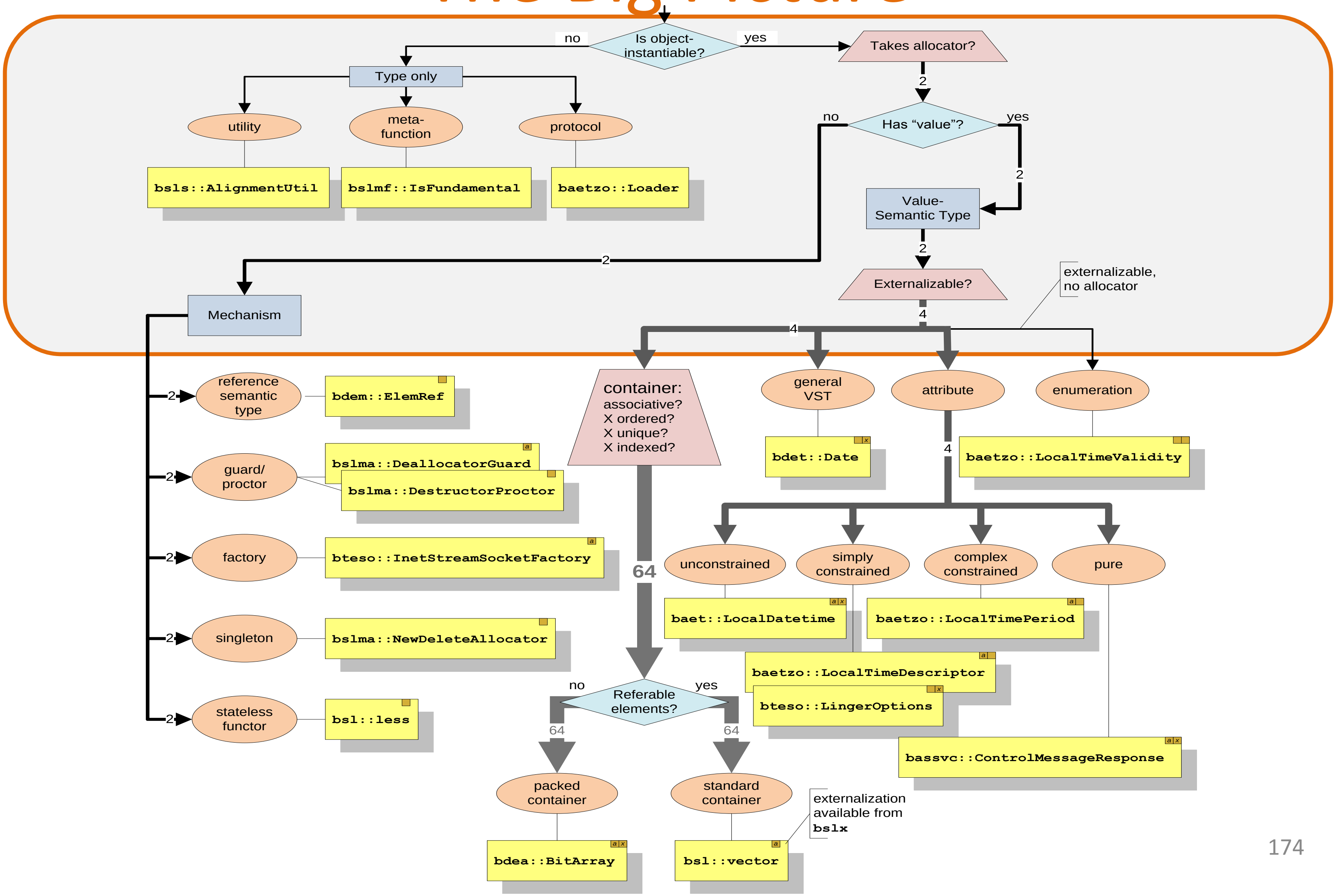
# Top-Level Categorizations



# Top-Level Categorizations

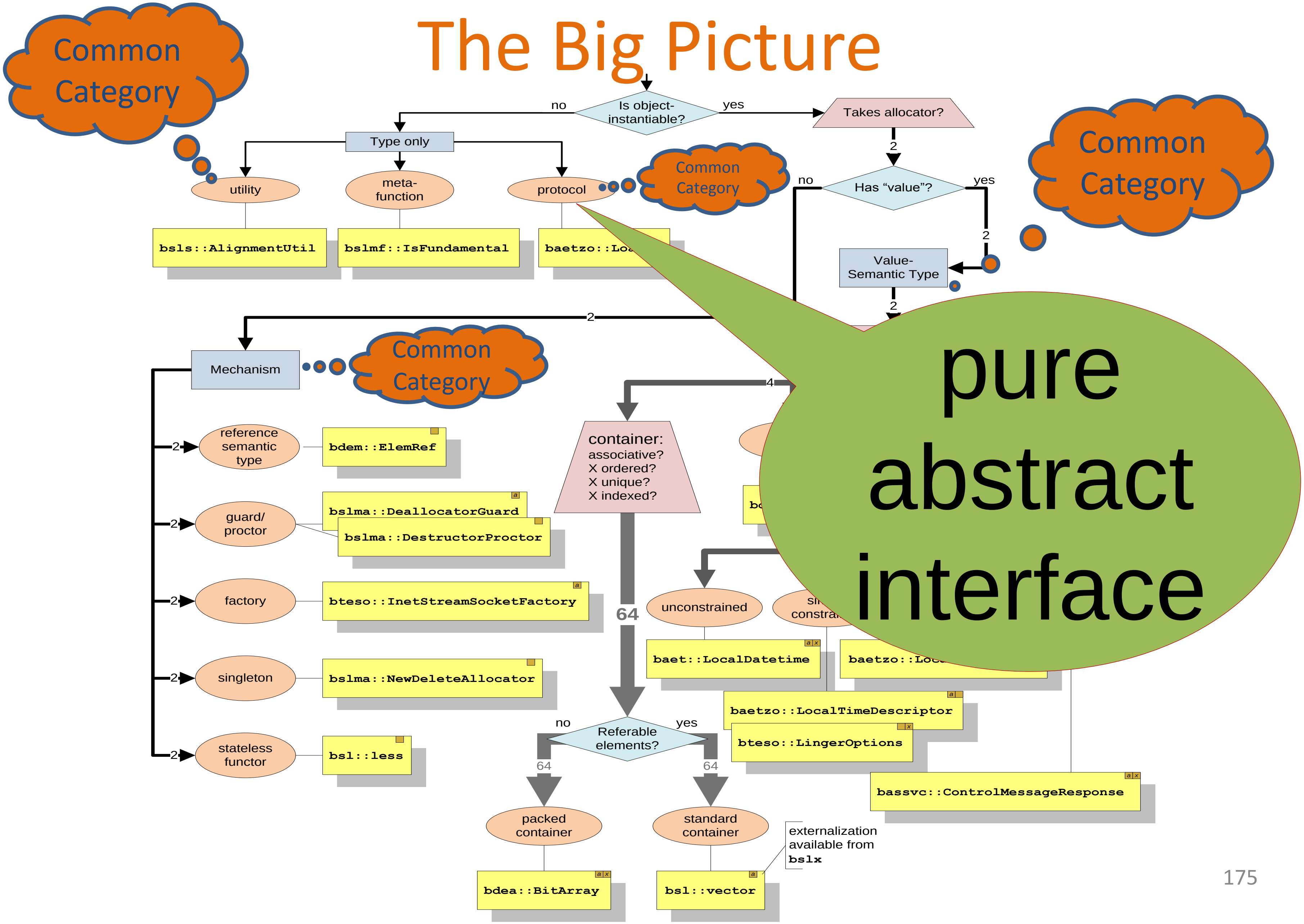


# The Big Picture



# 2. Understanding Value Semantics

# The Big Picture



# The Big Picture

## QUESTION:

What does it mean for two abstract types to compare equal?

# The Big Picture

## QUESTION:

What does it mean for two abstract types to compare equal?

# The Big Picture

## QUESTION:

~~What does it mean for two abstract types to compare equal?~~

Data members are for:

# "Variation in Value"

—Tom Cargill (c. 1992)

# What syntax should value types have?

# Value-Semantic Properties

A *value-semantic* type `T` defines the following:

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);

# Value-Semantic Properties

A *value-semantic* type `T` defines … ving:

> Typically, but not necessarily (e.g., `int`)

- Default construction: T **a**, **b**; assert(**a** == **b**);

# Value-Semantic Properties

A *value-semantic* type `T` defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);

Typically, but not necessarily (e.g., `int`)

However "zero" *initialization* `assert(T() == T());` Is `true`

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);
- Copy construction: T **a**, **b**(**a**); assert(**a** == **b**);

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);
- Copy construction: T **a**, **b**(**a**); assert(**a** == **b**);
- Destruction: *(Release all resources.)*

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);
- Copy construction: T **a**, **b**(**a**); assert(**a** == **b**);
- Destruction: *(Release all resources.)*
- Copy assignment: **a** = **b;** assert(**a** == **b**);

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**; assert(**a** == **b**);
- Copy construction: T **a**, **b**(**a**); assert(**a** == **b**);
- Destruction: *(Release all resources.)*
- Copy assignment: **a** = **b**; assert(**a** == **b**);
- Swap (if well-formed): T **a**($\alpha$), **b**($\beta$); swap(**a**, **b**);
  assert($\beta$ == **a**);
  assert($\alpha$ == **b**);

# Value-Semantic Properties

A *value-semantic* type T defines the following:

- Default construction: T **a**, **b**;
- Copy co
- ates.)
- a = b; assert(**a** == **b**);
- Swap (if well-formed): T **a**($\alpha$), **b**($\beta$); swap(**a**, **b**);

assert($\beta$ == **a**);

assert($\alpha$ == **b**);

**“Regular Type”**

# Value-Semantic Properties

`operator==(T, T)` describes what's called an *equivalence relation*:

1. ***a*** == ***a*** (reflexive)
2. ***a*** == ***b*** ⇔ ***b*** == ***a*** (symmetric)
3. ***a*** == ***b*** && ***b*** == ***c*** ⇒ ***a*** == ***c*** (transitive)

# Value-Semantic Properties

`operator==(T, T)` describes what's called an *equivalence relation*:

1. ***a*** == ***a*** (reflexive)
2. ***a*** == ***b*** ⬄ ***b*** == ***a*** (symmetric)
3. ***a*** == ***b*** && ***b*** == ***c*** ⇨ ***a*** == ***c*** (transitive)

➢ ! (***a*** == ***b***) ⬄ ***a*** != ***b***

# Value-Semantic Properties

`operator==(T, T)` describes what's called an *equivalence relation*:

1. ***a*** == ***a*** (reflexive)
2. ***a*** == ***b*** ⇔ ***b*** == ***a*** (symmetric)
3. ***a*** == ***b*** && ***b*** == ***c*** ⇒ ***a*** == ***c*** (transitive)

- ! (***a*** == ***b***) ⇔ ***a*** != ***b***
- **_a_ == _d_ (compiles) ⇔ _d_ == _a_ (compiles)**

  (Note that ***d*** is not of the same type as ***a***.)

# Value-Semantic Properties

`operator==(T,` ’s called

an *equivalence r*

1. ***a*** == ***a*** lexive)
2. ***a*** == ***b*** ⇔ ***b*** symmetric)
3. ***a*** == ***b*** && ***b*** == (transitive)

➢ `!(`***a*** `==` ***b***`)` ⇔ ***a*** `!=`

➢ ***a* == *d* (compiles) ⇔ *d* == *a* (compiles)**

(Note that ***d*** is <u>not</u> of the same type as ***a***.)

What am I talking about?

# Value-Semantic Properties

## *Member* `operator==`

```
class T {
    // ...
  public:
    // ...
    bool operator==(const T& rhs) const;
    // ...
};
```

# Value-Semantic Properties

## *Member* `operator==`

```
class T {
    // ...
  public:
    // ...
    bool operator==(const T& rhs) const;
    // ...
};
```

```
class D {
    // ...
  public:
    // ...
    operator const T&() const;
    // ...
};
```

# Value-Semantic Properties

## *Member* `operator==`

```
class T {
    // ...
  public:
    // ...
    bool operator==(const T& rhs) const;
    // ...
};
```

```
class D {
    // ...
  public:
    // ...
    operator const T&() const;
    // ...
};
```

```
void f(const T& a, const D& d)
{
    if (a == d) { /* … */ }

}
```

# Value-Semantic Properties

## *Member* `operator==`

```
class T {
    // ...
  public:
    // ...
    bool operator==(const T& rhs) const;
    // ...
};
```

Bad Idea

```
class D {
    // ...
  public:
    // ...
    operator const T&() const;
    // ...
};
```

```
void f(const T& a, const D& d)
{
    if (a == d) { /* … */ }
    if (d == a) { /* … */ }
}
```

# Value-Semantic Properties

## *Free* `operator==`

```
class T {
    // ...
  public:
    // ...
};
// ...
bool operator==(const T& lhs, const T& rhs);
```

```
class D {
    // ...
  public:
    // ...
    operator const T&() const;
    // ...
};
```

```
void f(const T& a, const D& d)
{
    if (a == d) { /* … */ }
}
```

# Value-Semantic Properties

## *Free* `operator==`

```
class T {
    // ...
  public:
    // ...
};
// ...
bool operator==(const T& lhs, const T& rhs);
```

**(proper)**

```
class D {
    // ...
  public:
    // ...
    operator const T&() const;
    // ...
};
```

```
void f(const T& a, const D& d)
{
    if (a == d) { /* … */ }
    if (d == a) { /* … */ }
}
```

# Value-Semantic Properties

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...

    // ...
};
// ...
```

# Value-Semantic Properties

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...


    // ...
};
// ...
bool operator==(const Str&   lhs, const Str& rhs);
```

# Value-Semantic Properties

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...

    // ...
};
// ...
bool operator==(const Str&   lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

# Value-Semantic Properties

*Member* Operator==

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
    bool operator==(const char *rhs) const;
    // ...
};
// ...
bool operator==(const Str&   lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

# Value-Semantic Properties

*Member* Operator==

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
    bool operator==(const char *rhs) const;
    // ...
};
// ...
bool operator==(const Str&   lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

# Value-Semantic Properties

*Member* Operator==

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
    bool operator==(const char *rhs) const;
    // ...
};
// ...
bool operator==(const Str&  lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

```
class Bar {
    // ...
  public:
    // ...
    operator const char *() const;
    // ...
};
```

# Value-Semantic Properties

```
class Str {          Member Operator==
    // ...
  public:
    Str(const char *other);
    // ...
    bool operator==(const char *rhs) const;
    // ...
};
// ...
bool operator==(const Str&   lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

```
void f(const Foo& foo, const Bar& bar)
{
    if (bar == foo) { /* ... */ }
}
```

```
class Bar {
    // ...
  public:
    // ...
    operator const char *() const;
    // ...
};
```

# Value-Semantic Properties

*Member* Operator==

Bad Idea

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
    bool operator==(const char *rhs) const;
    // ...
};
// ...
bool operator==(const Str&  lhs, const Str& rhs);
bool operator==(const char *lhs, const Str& rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

```
void f(const Foo& foo, const Bar& bar)
{
    if (bar == foo) { /* ... */ }
    if (foo == bar) { /* ... */ }
}
```

```
class Bar {
    // ...
  public:
    // ...
    operator const char *() const;
    // ...
};
```

# Value-Semantic Properties

*Free* Operator==

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
};
// ...
bool operator==(const Str&   lhs, const Str&   rhs);
bool operator==(const char  *lhs, const Str&   rhs);
bool operator==(const Str&   lhs, const char *rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

```
void f(const Foo& foo, const Bar& bar)
{
    if (bar == foo) { /* ... */ }
}
```

```
class Bar {
    // ...
  public:
    // ...
    operator const char *() const;
    // ...
};
```

# Value-Semantic Properties

***Free*** **Operator==**

**(proper)**

```
class Str {
    // ...
  public:
    Str(const char *other);
    // ...
};
// ...
bool operator==(const Str&   lhs, const Str&   rhs);
bool operator==(const char  *lhs, const Str&   rhs);
bool operator==(const Str&   lhs, const char *rhs);
```

```
class Foo {
    // ...
  public:
    // ...
    operator const Str&() const;
    // ...
};
```

```
void f(const Foo& foo, const Bar& bar)
{
    if (bar == foo) { /* … */ }
    if (foo == bar) { /* … */ }
}
```

```
class Bar {
    // ...
  public:
    // ...
    operator const char *() const;
    // ...
};
```

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

Same for *most** binary operators with `const` parameters:

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

Same for *most** binary operators with `const` parameters:

✓ **== !=** (equality)

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

Same for *most** binary operators with `const` parameters:

✓ `==` `!=` `(equality)`

✓ `<` `<=` `>` `=>` `(relational)`

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

Same for *most** binary operators with `const` parameters:

✓ `==` `!=` `(equality)`

✓ `<` `<=` `>` `=>` `(relational)`

✓ `+` `-` `*` `/` `%` `(arithmetic)`

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

Same for *most** binary operators with `const` parameters:

- ✓ `==` `!=` (equality)
- ✓ `<` `<=` `>` `=>` (relational)
- ✓ `+` `-` `*` `/` `%` (arithmetic)
- ✓ `|` `&` `^` `<<` `>>` (logical)

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

**But not `operator@=`**

✓ `==` `!=` (equality)

✓ `<` `<=` `>` `=>` (relational)

✓ `+` `-` `*` `/` `%` (arithmetic)

✓ `|` `&` `^` `<<` `>>` (logical)

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

**But not `operator@=`**

| | | | | | | |
|---|---|---|---|---|---|---|
| ✓ | `==` | `!=` | | | | `(equality)` |
| ✓ | `<` | `<=` | `>` | `=>` | | `(relational)` |
| ✓ | `+` | `-` | `*` | `/` | `%` | `(arithmetic)` |
| ✓ | `\|` | `&` | `^` | `<<` | `>>` | `(logical)` |
| ✗ | `+=` | `-=` | `*=` | `/=` | `%=` | `(assignment)` |

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# Value-Semantic Properties

The `operator==` should ***ALWAYS*** be free!

**But not `operator@=`**

| | | | | | | |
|---|---|---|---|---|---|---|
| ✓ | `==` | `!=` | | | | `(equality)` |
| ✓ | `<` | `<=` | `>` | `=>` | | `(relational)` |
| ✓ | `+` | `-` | `*` | `/` | `%` | `(arithmetic)` |
| ✓ | `\|` | `&` | `^` | `<<` | `>>` | `(logical)` |
| ✗ | `+=` | `-=` | `*=` | `/=` | `%=` | `(assignment)` |
| ✗ | `\|=` | `&=` | `^=` | `<<=` | `>>=` | `(assignment)` |

*Except for operators such as `operator[]` that return a ***reference*** instead of a ***value***, and `operator()`.

# What semantics should value-type operations have?

# Where is “Value” Defined?

# Where is “Value” Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` …

# Where is “Value” Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T`

1. Derive from the physical state of *only* that instance of `T`.

# Where is "Value" Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T`

1. Derive from the physical state of *only* that instance of `T`.
2. Must *respectively* "have" (refer to) ***the same*** value in order for two instances of `T` to have (refer to) ***the same*** value *as a whole*.

# Where is “Value” Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` that

1. Derive from the physical state of *only* that instance of `T`.

2. Must *respectively* “have” (refer to) *the same* value in order for two instances of `T` to have (refer to) *the same* value *as a whole*.

# Where is "Value" Defined? `Copy Constructor?`

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` that

1. Derive from the physical state of *only* that instance of `T`.
2. Must *respectively* "have" (refer to) *the same* value in order for two instances of `T` to have (refer to) *the same* value *as a whole*.

# Where is "Value" Defined?

# `Copy Constructor?`

- By def., *all salient attributes* <u>must</u> be copied.

# Where is "Value" Defined?
# `Copy Constructor?`

- By def., *all salient attributes* <u>must</u> be copied.
- What about "non-salient" attributes?
  - E.g., `capacity()`

# Where is "Value" Defined? `Copy Constructor?`

- By def., *all salient attributes* <u>must</u> be copied.
- What about "non-salient" attributes?
  - E.g., `capacity()`
- Non-salient attributes ***may*** or ***may not*** be copied.

**More on this later...**

# Where is "Value" Defined?

# `Copy Constructor?`

- By def., *all salient attributes* <u>must</u> be copied.
- What about "non-salient" attributes?
  - E.g., `capacity()`
- Non-salient attributes ***may*** or ***may not*** be copied.
- Hence, we ***cannot*** infer from the *implementation* of a Copy Constructor which attributes are "salient."

# Where is "Value" Defined?
# `Copy Constructor?`

- By def., *all salient attributes* must be copied.
- What about "non-salient" attributes?
  - E.g., `capacity()`
- Non-salient attributes *may* or *may not* be copied.
- Hence, we ***cannot*** infer from the *implementation* of a Copy Constructor which attributes are "salient."
- ***Cannot*** tell us if two objects have *the same* value!

# Where is "Value" Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` that

1. Derive from the physical state of *only* that instance of `T`.

2. Must *respectively* "have" (refer to) *the same* value in order for two instances of `T` to "have" (refer to) *the same* value *as a whole*.

# Where is “Value” Defined?

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` that

1. Derive from the physical state of *only* that instance of `T`.

2. Must *respectively* compare equal in order for two instances of `T` to compare equal *as a whole*.

# Where is “Value” Defined?

# `operator==`

The ***salient attributes*** of a type `T` are the documented set of named attributes whose respective values for a given instance of `T` that

1. Derive from the physical state of *only* that instance of `T`.

2. Must *respectively* compare equal in order for two instances of `T` to compare equal *as a whole*.

# Where is "Value" Defined?
# `operator==`

The associated, homogeneous (free) `operator==` for a type `T`

# Where is "Value" Defined?

## `operator==`

The associated, homogeneous (free) `operator==` for a type `T`

**Implementation**

1. **Provides** an ***operational definition*** of what it means for two objects of type `T` to have "the same" ***value***.

# Where is "Value" Defined?
# `operator==`

The associated, homogeneous (free) `operator==` for a type `T`

**Implementation**

1. **Provides** an ***operational definition*** of what it means for two objects of type `T` to have "the same" ***value***.

2. **Defines** the ***salient attributes*** of `T` as those attributes whose respective values must compare equal in order for two instances of `T` to compare equal.

**Interface/Contract**

# Value-Semantic Properties

Value-semantic objects share many properties.

# Value-Semantic Properties

Value-semantic objects share many properties.

- Each of these properties is objectively verifiable, irrespective of the intended application domain.

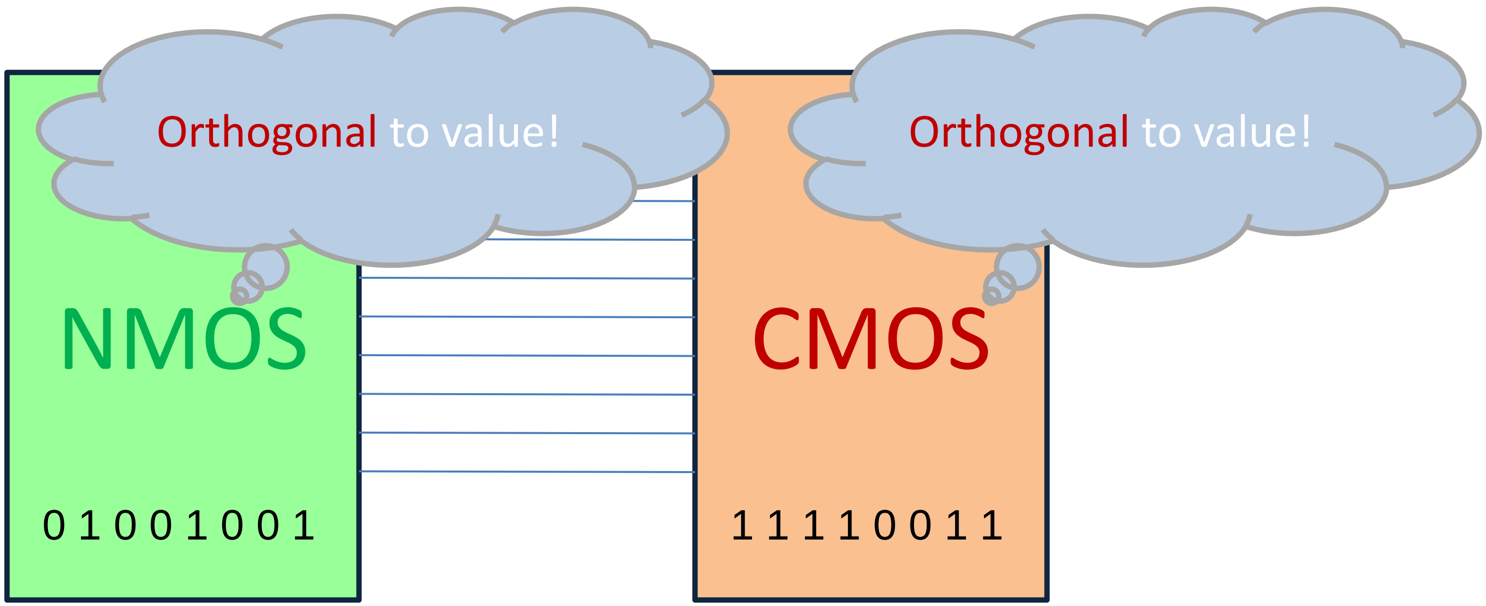
# Value-Semantic Properties

Value-semantic objects share many properties.

- Each of these properties is objectively verifiable, irrespective of the intended application domain.
- Most are (or should be) intuitive.

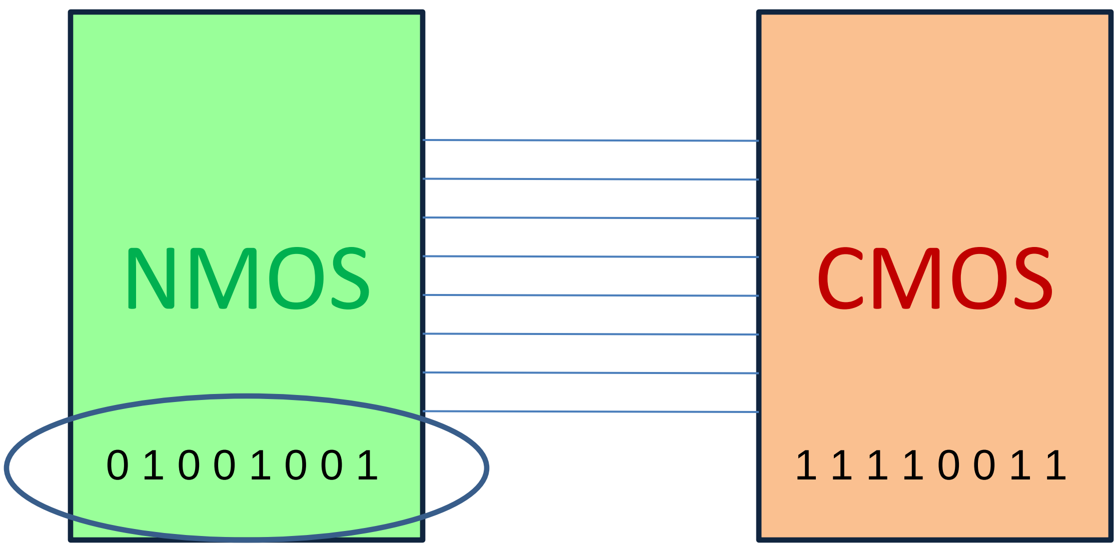
# What should be copied?

Should attributes that are orthogonal to value be copied?



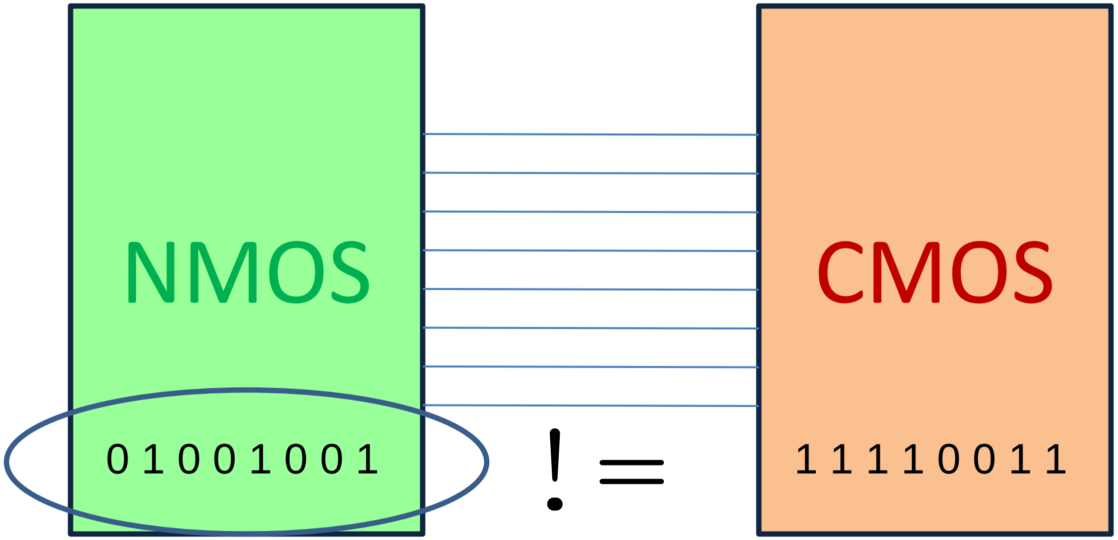
# What should be copied?

Should attributes that are orthogonal to value be copied?



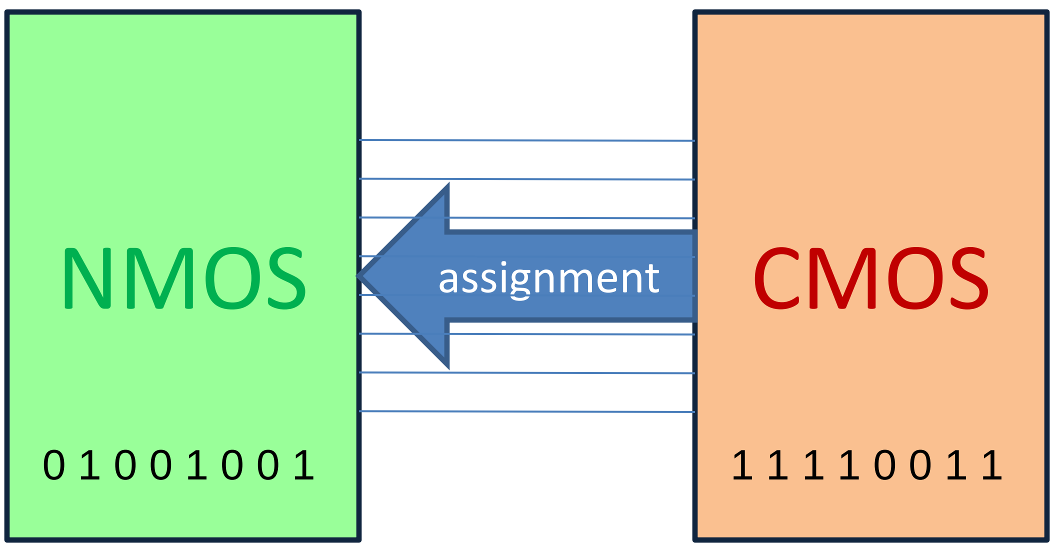
# What should be copied?

Should attributes that are orthogonal to value be copied?



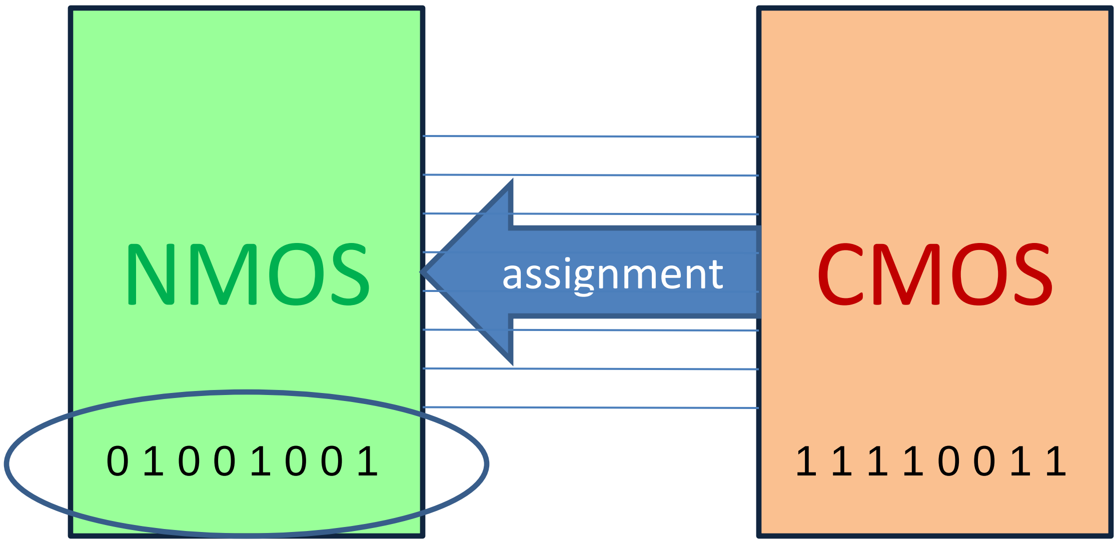
# What should be copied?

Should attributes that are orthogonal to value be copied?



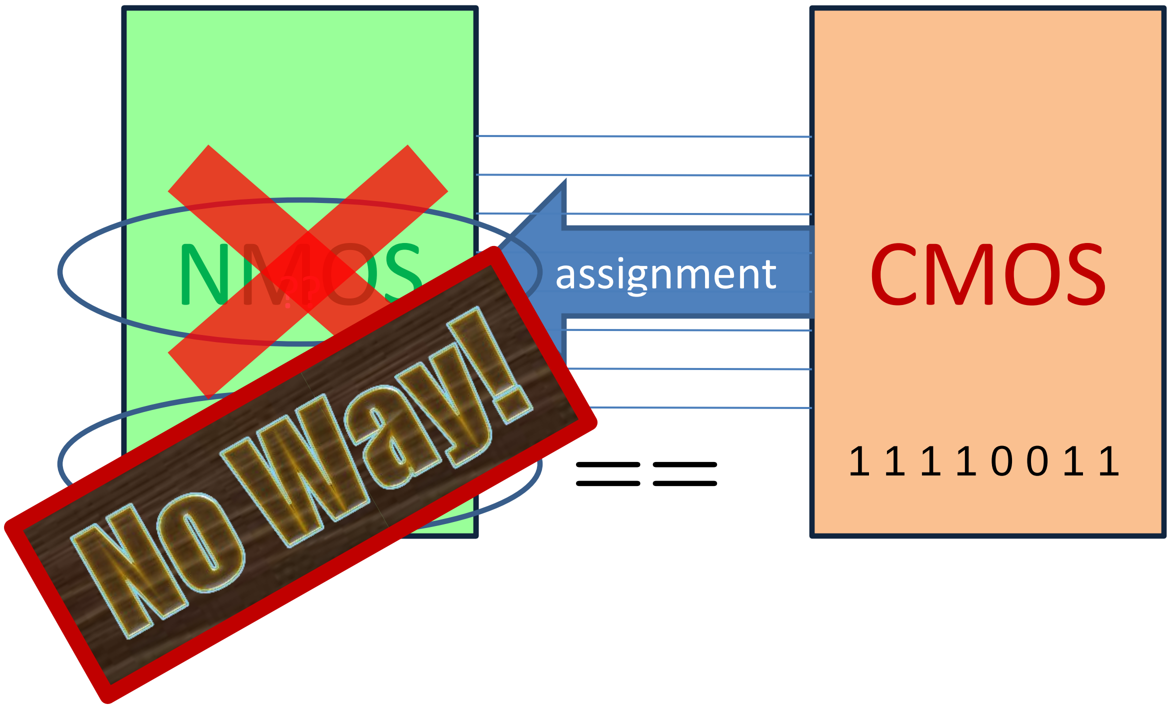
# What should be copied?

Should attributes that are orthogonal to value be copied?



# What should be copied?

Should attributes that are orthogonal to value be copied?



# What should be copied?

**V-TABLE POINTER?**

**ALLOCATOR?**

# Value-Semantic Properties

## Selecting Salient Attributes

As it turns out...

**Choosing salient attributes <u>appropriately</u> *will* affect our ability to test <u>thoroughly</u>.**

# Value-Semantic Properties

If `T` is a value-semantic type,

***a***, ***b***, and ***c*** are objects of type `T`, and

***d*** is an object of some *other* type `D`, then

# Value-Semantic Properties

If `T` is a value-semantic type,

***a***, ***b***, and ***c*** are objects of type `T`, and

***d*** is an object of some *other* type `D`, then

- ***a*** == ***b*** ⇔ ***a*** and ***b*** have the same *value* (assuming an associated `operator==` exists).

# Value-Semantic Properties

If `T` is a value-semantic type,

***a***, ***b***, and ***c*** are objects of type `T`, and

***d*** is an object of some *other* type `D`, then

- ***a*** == ***b*** ⇔ ***a*** and ***b*** have the same *value* (assuming an associated `operator==` exists).

# Value-Semantic Properties

If `T` is a value-semantic type,

*a*, *b*, and *c* are objects of type `T`, and

*d* is an object of some *other* type `D`, then

- ***a*** == ***b*** ⇔ ***a*** and ***b*** have the <u>same *value*</u> (assuming an associated `operator==` exists).

**(Sometimes a value-semantic type is "almost" *regular.*)**

# Value-Semantic Properties

If `T` is a value-semantic type,

***a***, ***b***, and ***c*** are objects of type `T`, and

***d*** is an object of some *other* type `D`, then

- ***a*** == ***b*** ⇔ ***a*** and ***b*** have the same *value* (assuming an associated `operator==` exists).

(Sometimes a value-semantic type is "almost" regular.)

MORE ON THIS LATER

# Value-Semantic Properties

If `T` is a value-semantic type,

***a***, ***b***, and ***c*** are objects of type `T`, and

***d*** is an object of some *other* type `D`, then

- ***a*** `==` ***b*** ⇔ ***a*** and ***b*** have the <u>same *value*</u> (assuming an associated `operator==` exists).
- The *value* of ***a*** is <u>independent</u> of *any* external object or state; any change to ***a*** must be accomplished via ***a***'s (public) interface.

# Value-Semantic Properties

Suppose a “*value-semantic*” object refers to another autonomous object in memory:

# Value-Semantic Properties

Suppose a “*value-semantic*” object refers to another autonomous object in memory:

```
class ElemPtr {
    Record *d_record_p;
    int     d_elementIndex;
  public:
    // …
};
```

# Value-Semantic Properties

Suppose a "*value-semantic*" object refers to another autonomous object in memory:

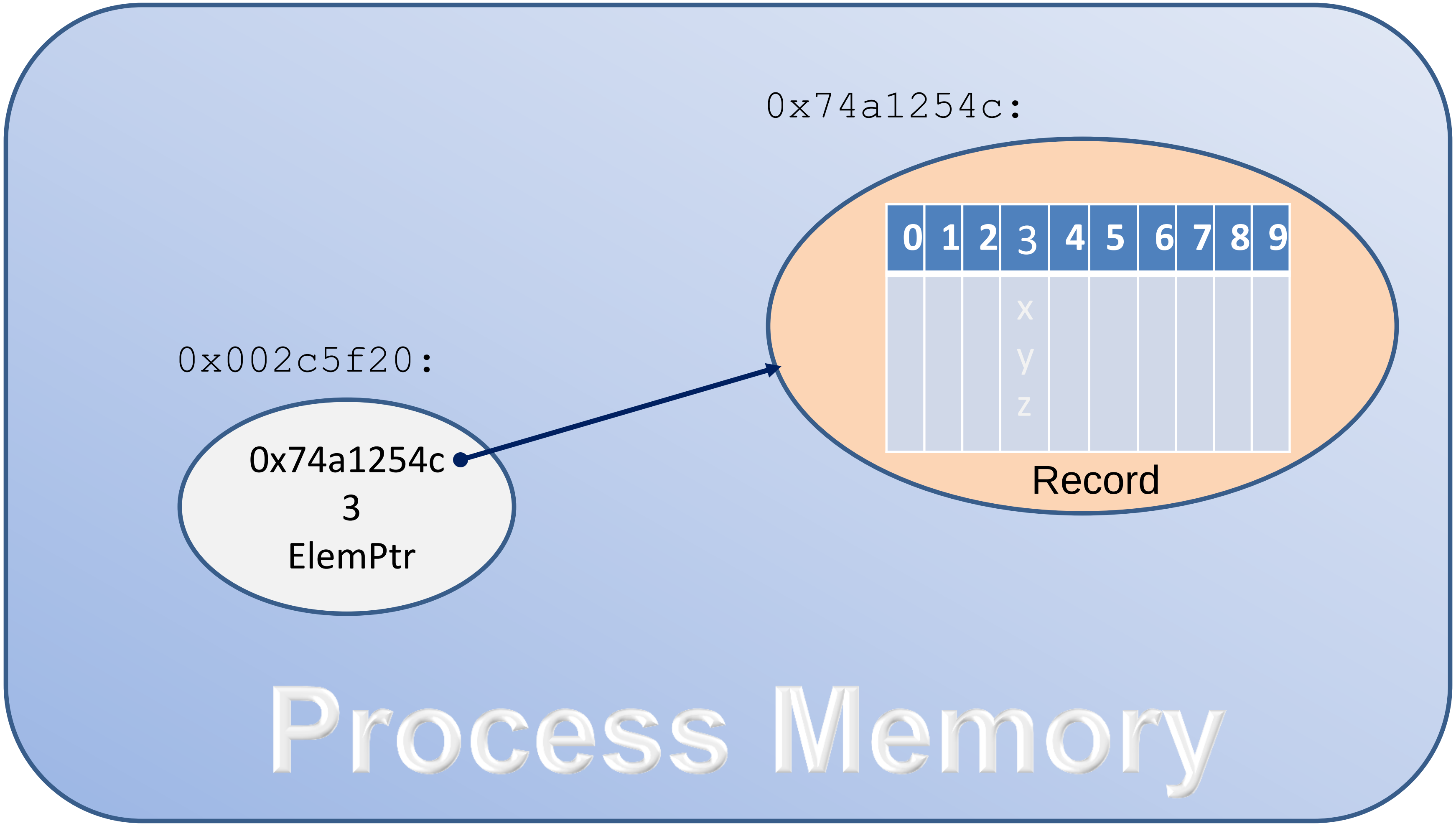
```
class ElemPtr {
    Record *d_record_p;
    int     d_elementIndex;
  public:
    // …
};
```

# Value-Semantic Properties

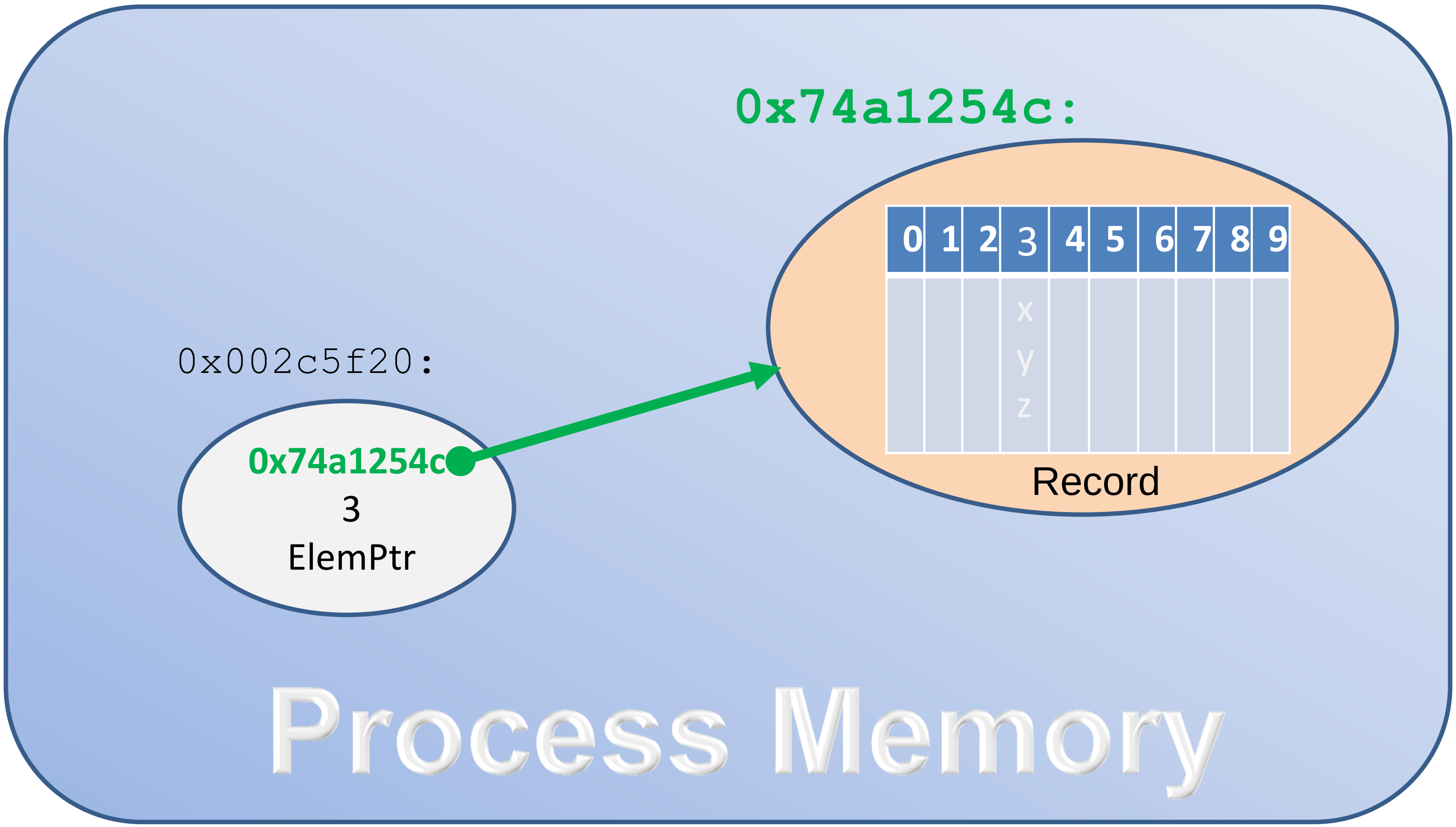
Suppose a "*value-semantic*" object refers to another autonomous object in memory:

```
class ElemPtr {
    Record *d_record_p;
    int     d_elementIndex;
  public:
    // …
};
```

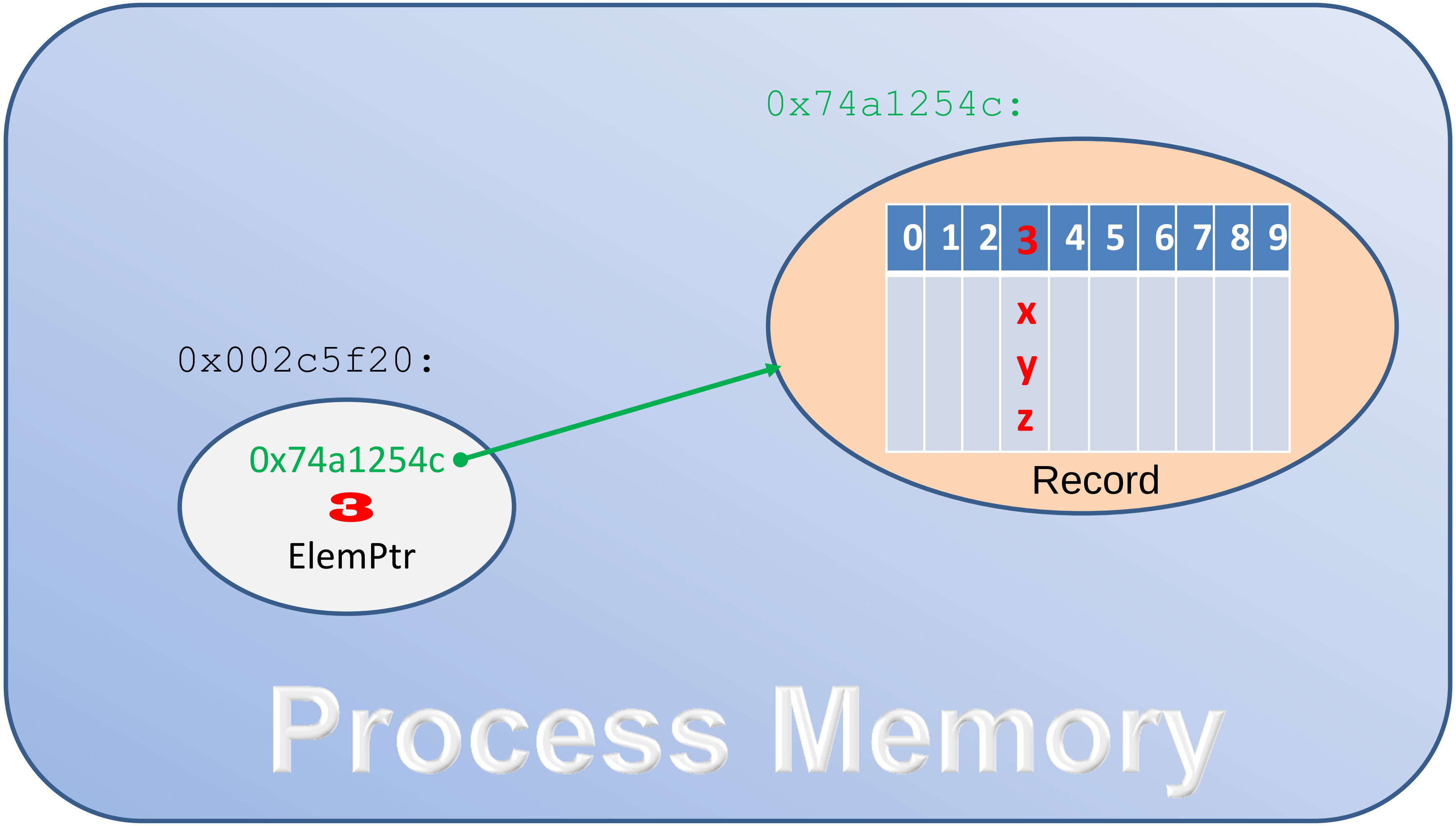
# Value-Semantic Properties



# Value-Semantic Properties



# Value-Semantic Properties



# Value-Semantic Properties

```
bool operator==(const ElemPtr& lhs,
                const ElemPtr& rhs);
```

# Value-Semantic Properties

```
bool operator==(const ElemPtr& lhs,
                const ElemPtr& rhs);
    // Two 'ElemPtr' objects have the
    // same value if they …
```

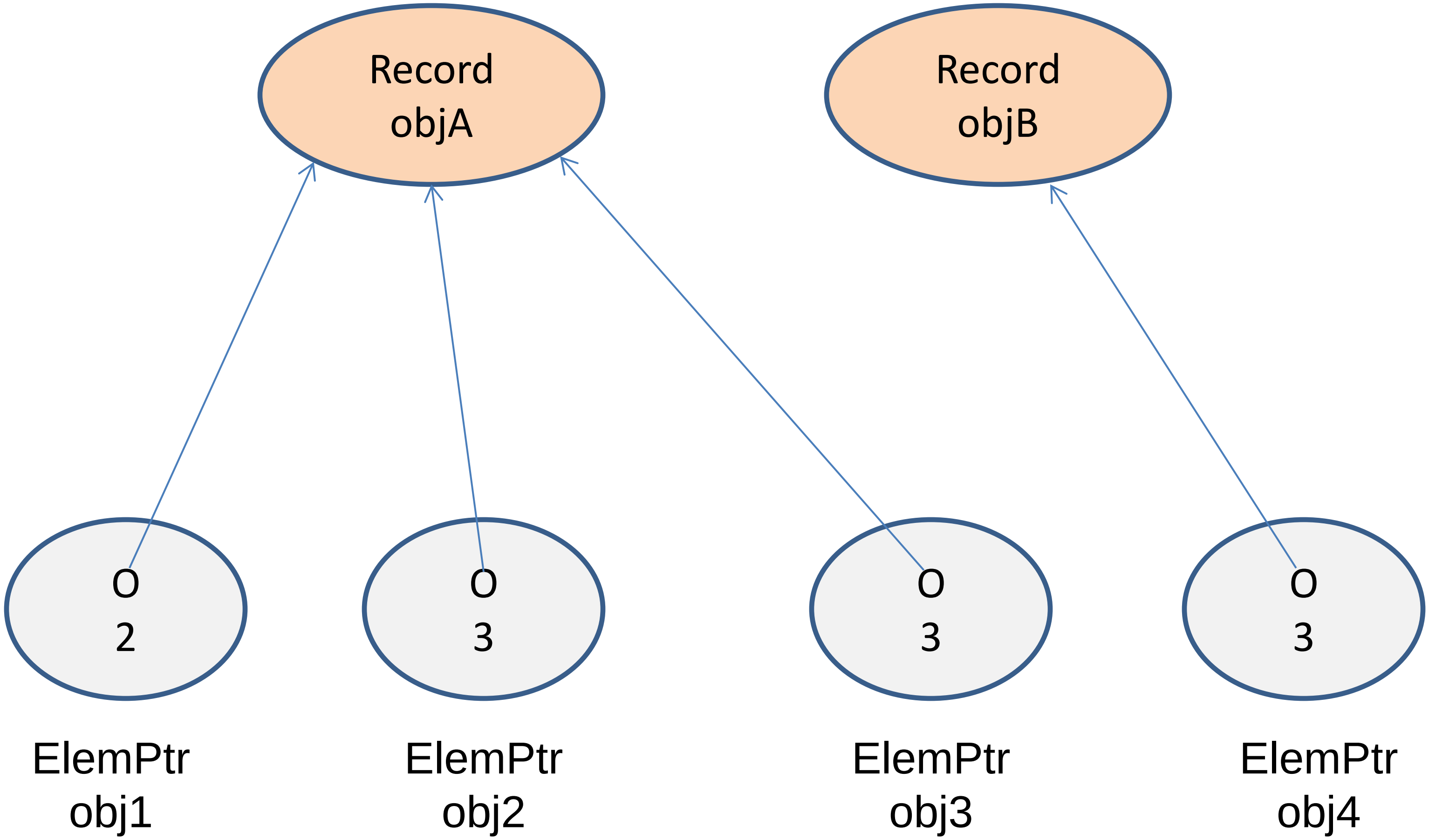
# Value-Semantic Properties

```
bool operator==(const ElemPtr& lhs,
                const ElemPtr& rhs);
    // Two 'ElemPtr' objects have the
    // same value if they (1) refer
    // to the same 'Record' object
    // (in the current process) …
```

# Value-Semantic Properties

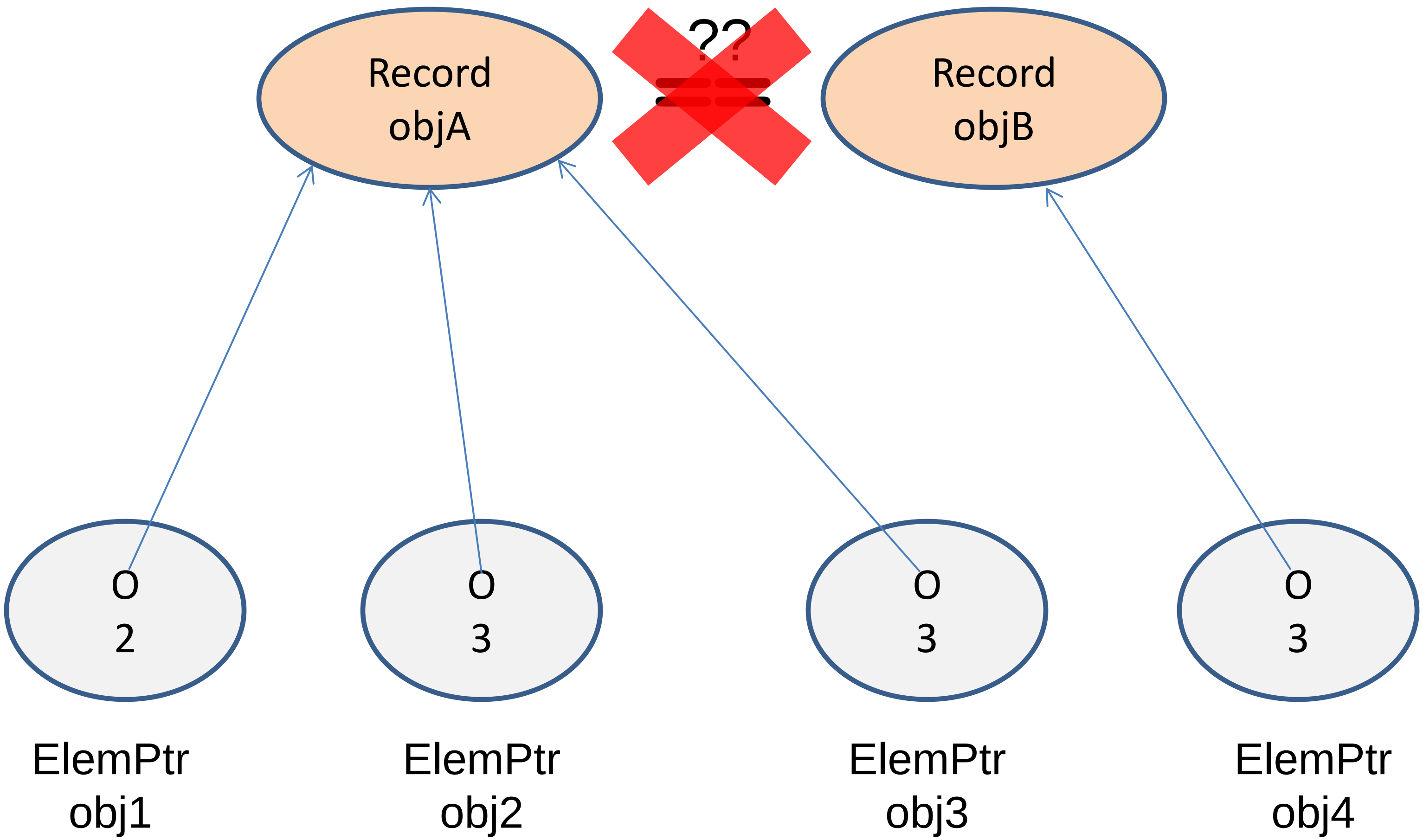
```
bool operator==(const ElemPtr& lhs,
                const ElemPtr& rhs);
    // Two 'ElemPtr' objects have the
    // same value if they (1) refer
    // to the same 'Record' object
    // (in the current process), and
    // (2) have the same element
    // index.
```

# Value-Semantic Properties

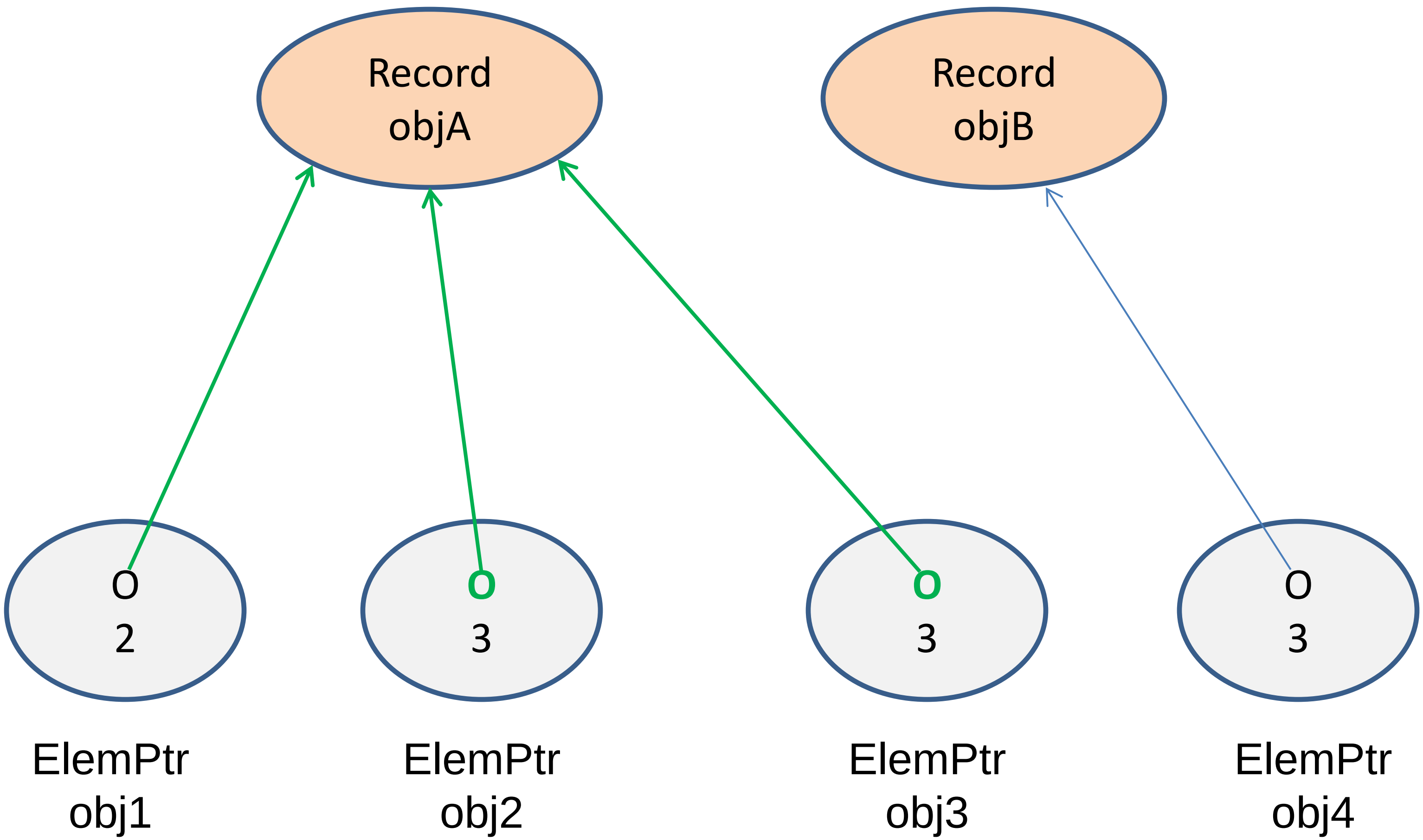


## 2. Understanding Value Semantics

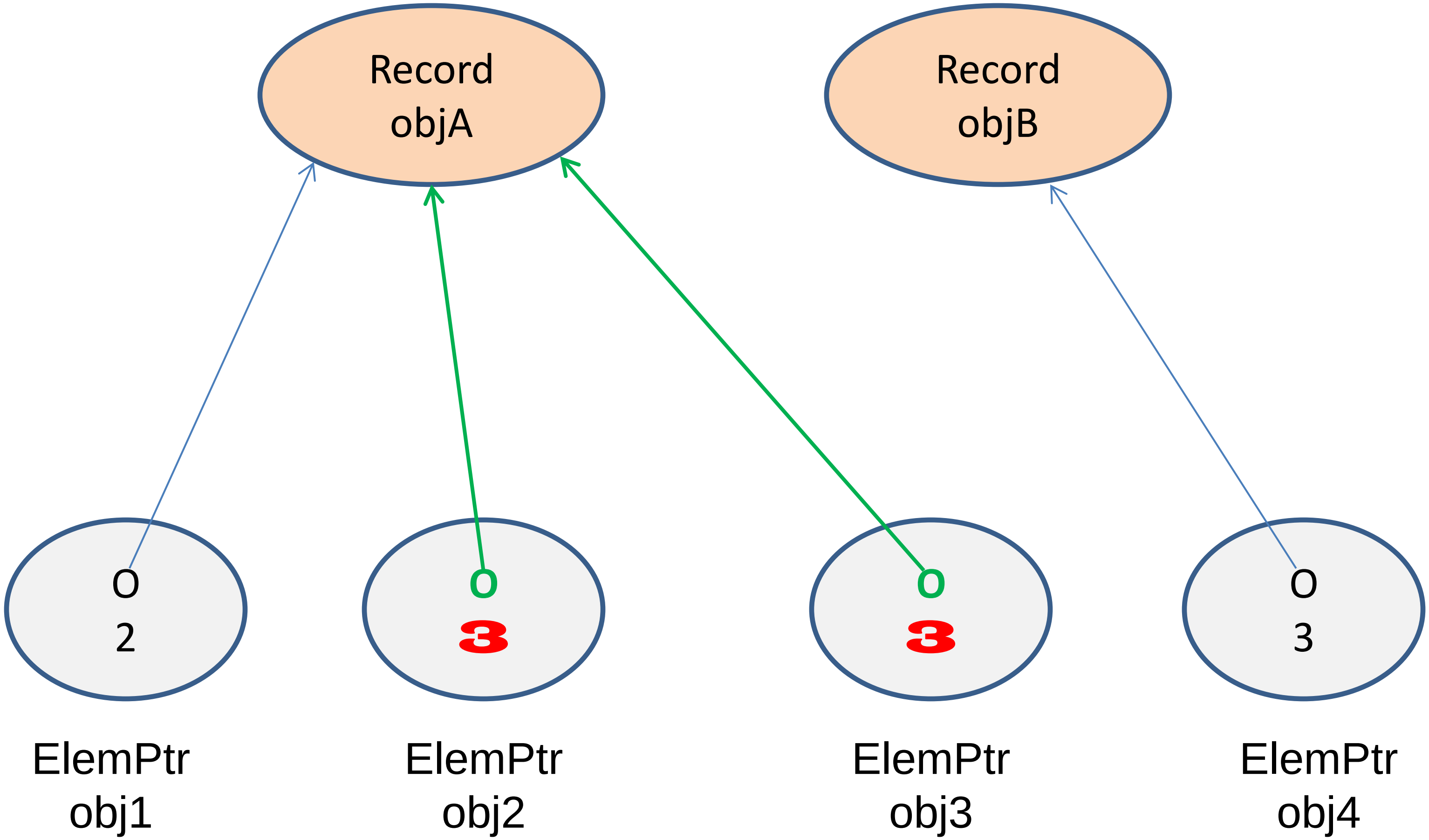
# Value-Semantic Properties



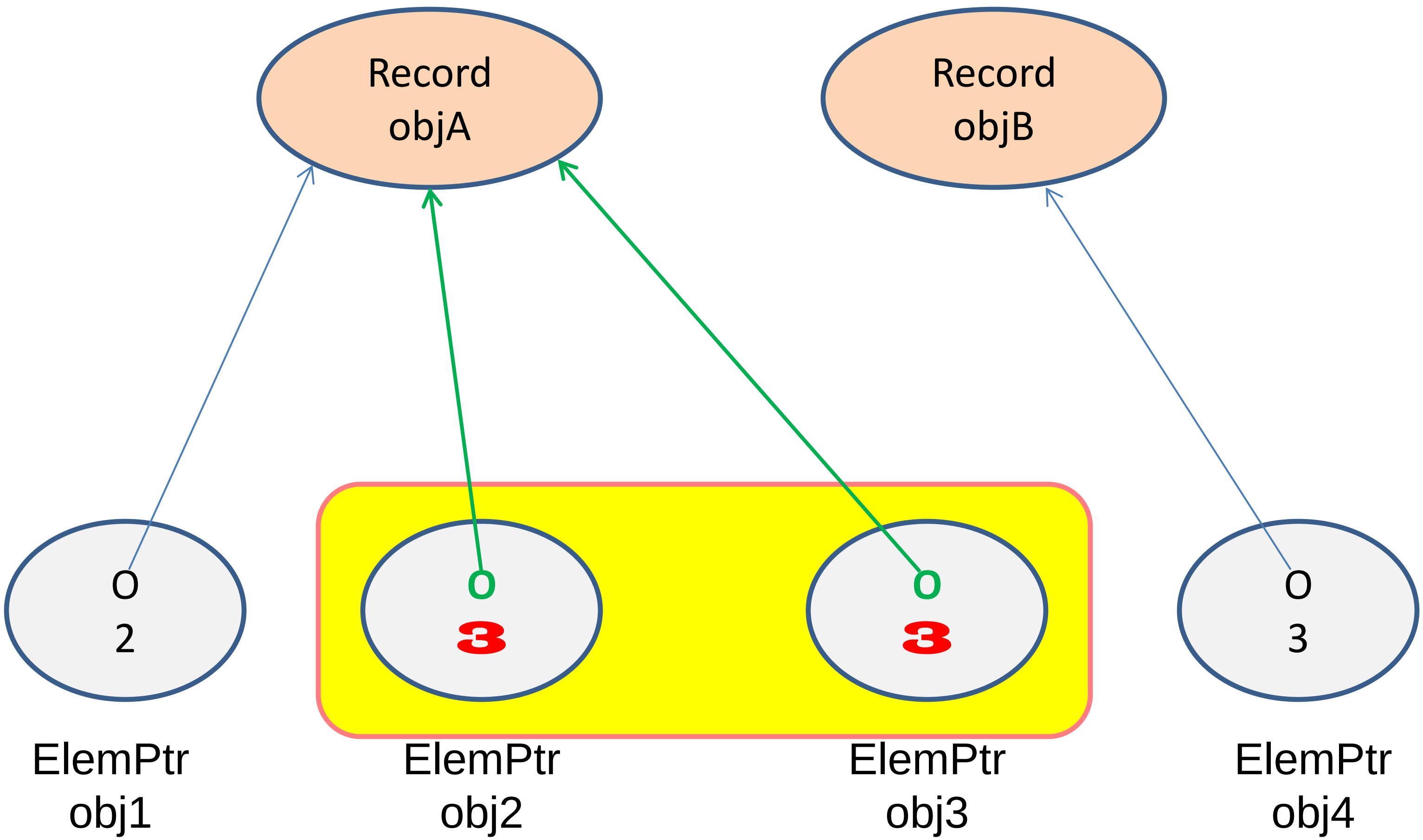
# Value-Semantic Properties



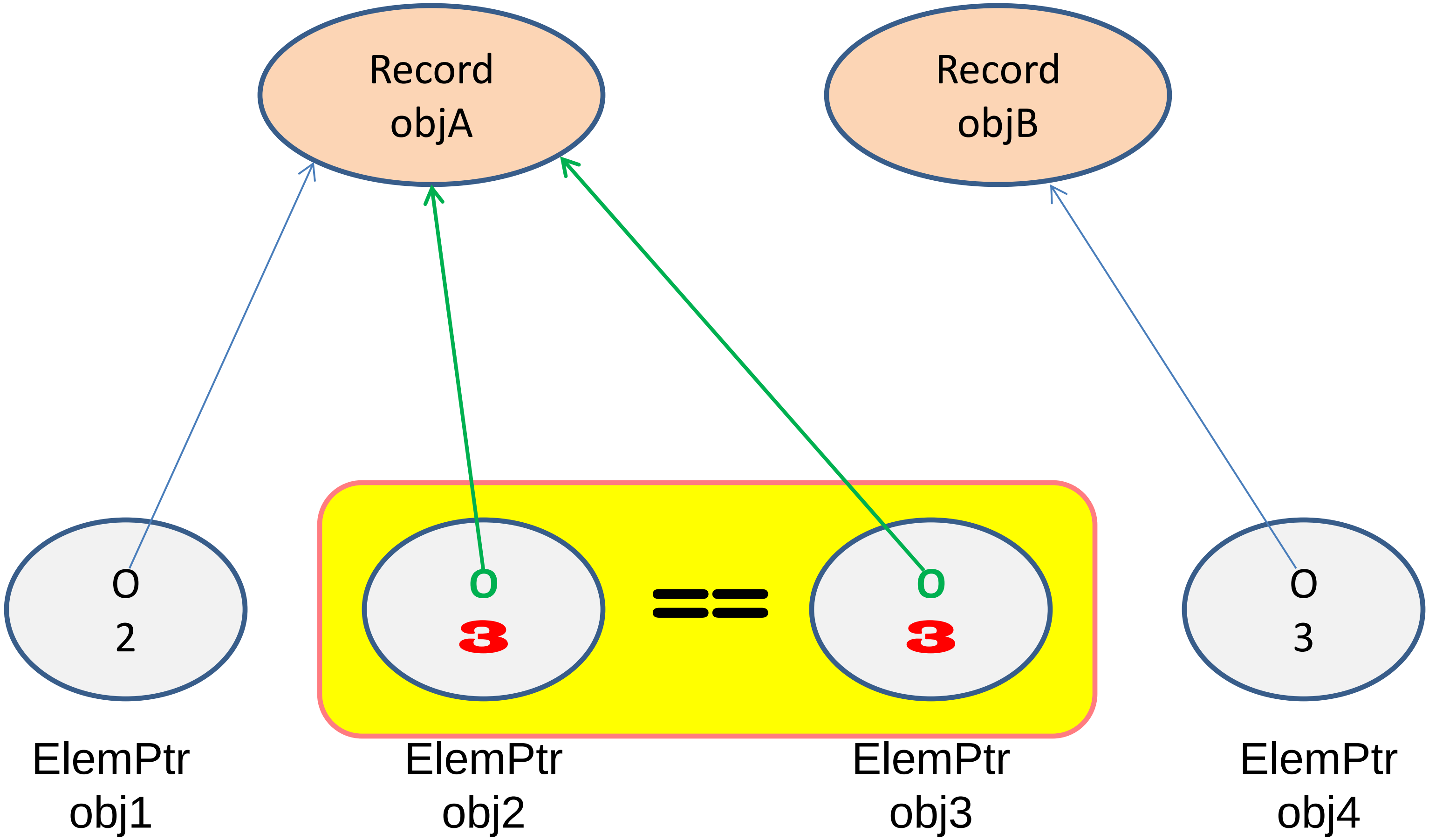
# Value-Semantic Properties



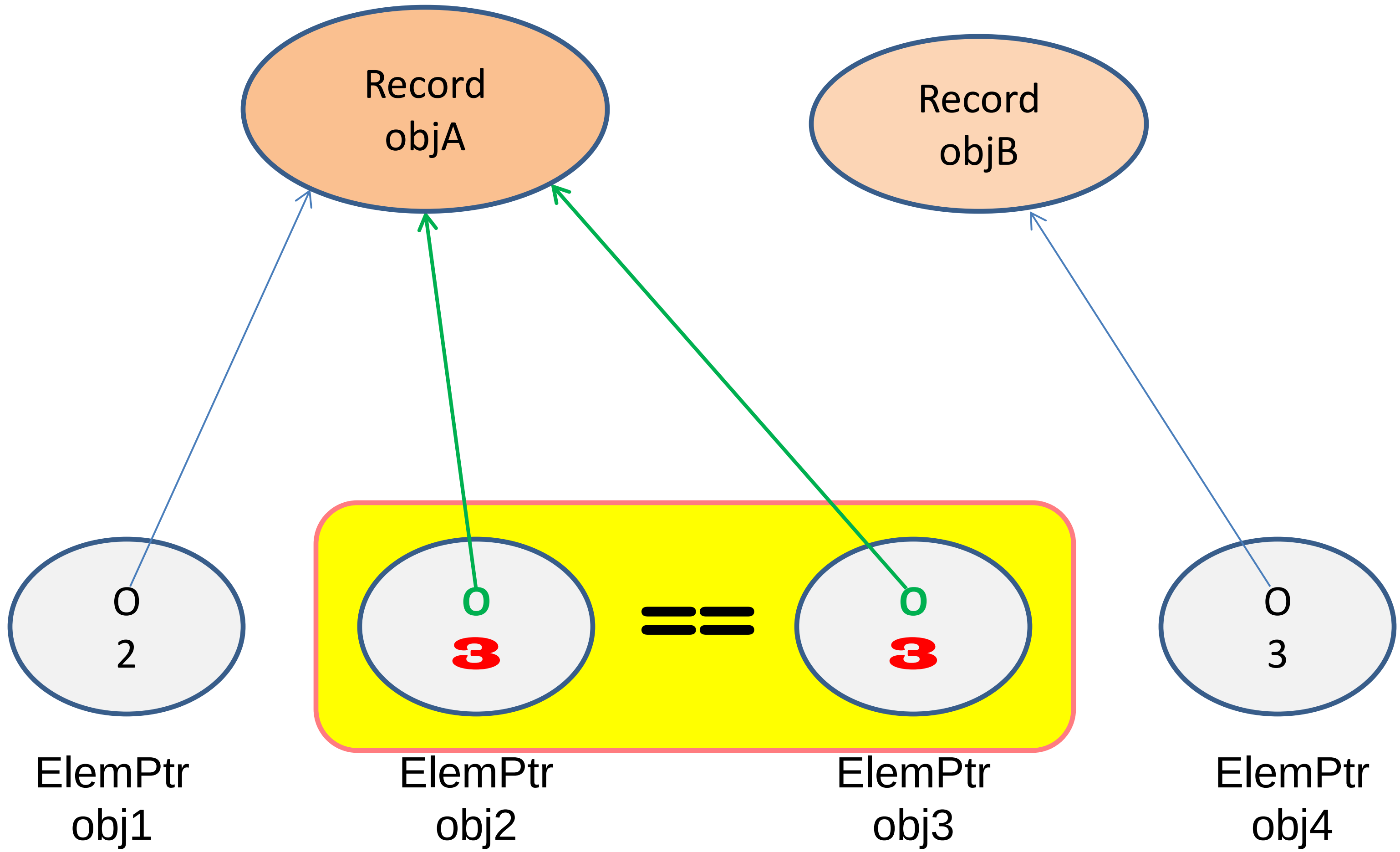
# Value-Semantic Properties



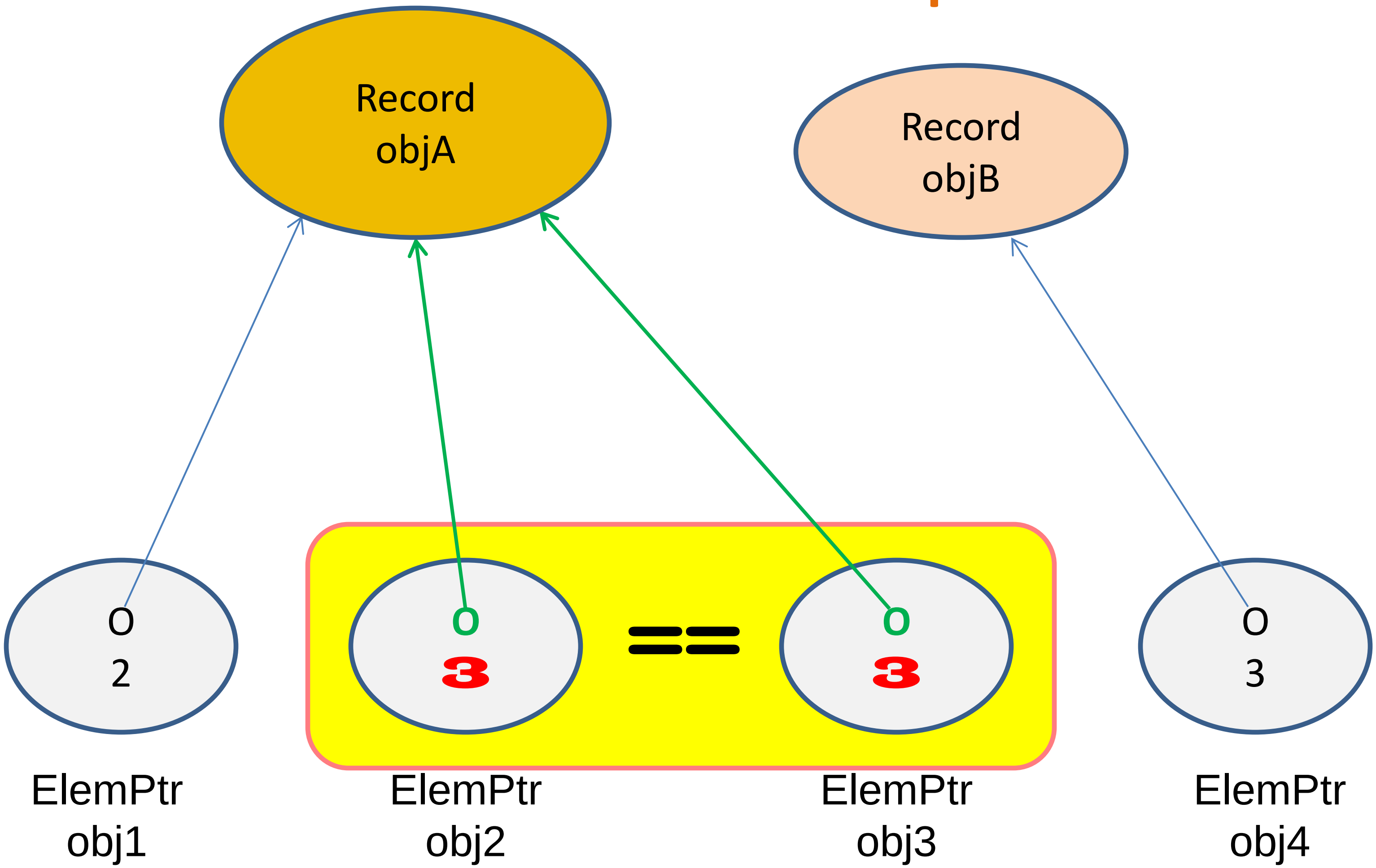
# Value-Semantic Properties



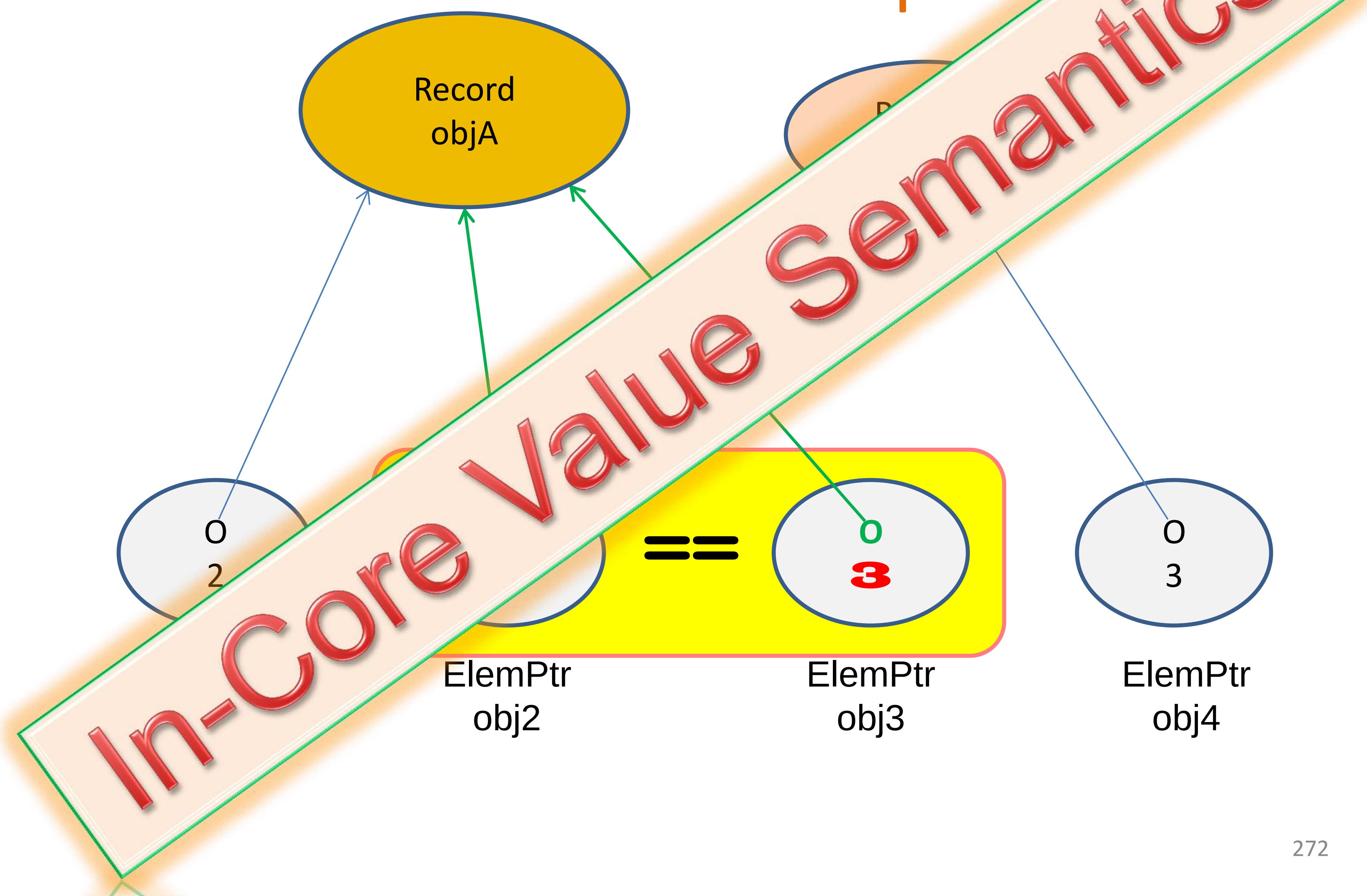
# Value-Semantic Properties



# Value-Semantic Properties



# Value-Semantic Properties



# Value-Semantic Properties

I.E., NOT FULL VALUE SEMANTICS

In-Co... ...ntics

ElemPtr
obj2

obj3

# Value-Semantic Properties

**In-Core Value Semantics, While Important, Is <u>Not</u> The Focus of Today's Talk**

Note that if we *were* to ascribe a notion of value to, say, a *scoped guard*, it would clearly be in-core only.

# “Value Types” having Value Semantics

# "Value Types" having Value Semantics

A C++ type that "properly" represents (a subset of) the values of an abstract "mathematical" type is said to have ***value semantics***.

# “Value Types” having Value Semantics

A C++ type that “properly” represents (a subset of) the values of an abstract “mathematical” type is said to have ***value semantics***.

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same **value** might not* exhibit "the same" ***observable behavior***.

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same* ***value*** *might not* exhibit "the same" ***observable behavior***.

E.g., one *might* allocate memory on an `append` operation, whereas another *might not.*

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same* ***value*** *might* *<u>not</u>* exhibit “the same” ***observable behavior***.

**HOWEVER**

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same **value** might <u>not</u>* exhibit "the same" ***observable behavior***.

HOWEVER

1. If **a** and **b** initially have the <u>*same value*</u>, and

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same* ***value*** *might* *not* exhibit "the same" ***observable behavior***.

HOWEVER

1. If **a** and **b** initially have the *same value*, and
2. the same operation is applied to each object, then

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same **value** might <u>not</u>* exhibit "the same" ***observable behavior***.

HOWEVER

1. If **a** and **b** initially have the <u>*same value*</u>, and
2. the <u>same operation</u> is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)

# Value-Semantic Properties

Recall that two *distinct* objects **a** and **b** of type T that have *the same* ***value*** *might* *<u>not</u>* exhibit "the same" ***observable behavior***.

HOWEVER

1. If **a** and **b** initially have the *<u>same value</u>*, and
2. the <u>same operation</u> is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will <u>again</u> have the *<u>same value</u>*!**

# Value-Semantic Properties

**There is a lot more to this story!**

Deciding what is (not) *salient* is surprisingly important.

*SUBTLE* ESSENTIAL PROPERTY OF *VALUE*

1. If **a** and **b** initially have the *same value*, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

# Value-Semantic Properties

That is…

```
if (a == b) {
        op1(a); op1(b); assert(a == b);
        op2(a); op2(b); assert(a == b);
        op3(a); op3(b); assert(a == b);
        op4(a); op4(b); assert(a == b);
           ⋮       ⋮              ⋮
}
```

**SUBTLE ESSENTIAL PROPERTY OF VALUE**

# Value-Semantic Pro

Note that this is <u>not</u> a test case, but rather a requirements specification.

That is…

```
if (a == b) {
        op1(a); op1(b); assert(a == b);
        op2(a); op2(b); assert(a == b);
        op3(a); op3(b); assert(a == b);
        op4(a); op4(b); assert(a == b);
           ⋮        ⋮               ⋮
}
```

**SUBTLE ESSENTIAL PROPERTY OF VALUE**

# Value-Semantic Properties

## QUESTION:

Suppose we have a “home grown” ordered-set type that can be initialized to a sequence of elements in either increasing or decreasing order:

```
template <class T>
class OrderedSet {
   // …
   OrderedSet(bool decreasingFlag = false);
   // …
};
```

# Value-Semantic Properties

## QUESTION:

Suppose we have a “home grown” ordered-set type that can be initialized to a sequence of elements in either increasing or decreasing order:

```
template <class T>
class OrderedSet {
   // …
   OrderedSet(bool decreasingFlag = false);
   // …
};
```

What if the two sets were constructed differently.

# Value-Semantic Properties

## QUESTION:

Suppose we have a “home grown” ordered-set type that can be initialized to a sequence of elements in either increasing or decreasing order:

```
template <class T>
class OrderedSet  {
   // …
   OrderedSet(bool decreasingFlag = false);
   // …
};
```

What if the two sets were constructed differently. Should any two empty objects be considered “equal”?

# Value-Semantic Properties

1. If **a** and **b** initially have the *same value*, and
2. the same operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

# Value-Semantic Properties

1. If **a** and **b** initially have the *same value*, and
2. the same ^salient operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

# Value-Semantic Properties

By *salient* we mean operations that directly reflect those in the mathematical type this C++ type is attempting to approximate.

1. If **a** and **b** initially have the *same value*, and
2. the same ^salient operation is applied to each object, then
3. (absent any *exceptions* or *undefined behavior*)
4. **both objects will again have the *same value*!**

# Value-Semantic Properties

## QUESTION:

What makes two unordered containers represent the same value?

Think about a bag of Halloween candy.

# Value-Semantic Properties

Note that this essential property applies <u>only</u> to objects of the ***same type***:

```
int x = 5;      int y = 5;   assert(x == y);
 x *= x;         y *= y;      assert(x == y);
 x *= x;         y *= y;      assert(x == y);
 x *= x;         y *= y;      assert(x == y);
     ⋮               ⋮                    ⋮
```

# Value-Semantic Properties

Note that this essential property applies only to objects of the ***same type***:

```
int x = 5;  short y = 5;   assert(x == y);
  x *= x;        y *= y;     assert(x == y);
  x *= x;        y *= y;     assert(x == y);
  x *= x;        y *= y;     assert(x == y);
    :               :                  :
```

Undefined Behavior!

# How do we design proper value types?

# Value-Semantic Properties

## Selecting Salient Attributes

```
class Rational {
    int d_numerator;
    int d_denominator;
  public:
    //
    int numerator() const;
    int denominator() const;
 };
 // …
 bool operator==(const Rational& lhs,
                 const Rational& rhs);
```

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

`numerator()/denominator()`

as the salient attribute?

# Value-Semantic Properties

## Selecting Salient Attributes

?

What about

`numerator()/denominator()`

as the salient attribute?

```
bool operator==(const Rational& lhs,
                const Rational& rhs);
```

// Two 'Rational' objects have the same value if
// the **ratio** of the values of '**numerator()**' and
// '**denominator ()**' for 'lhs' is the same as that for 'rhs'.

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

`numerator()/denominator()`

as <u>the</u> salient attribute?

$$\frac{1}{2} == \frac{2}{4} ?$$

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

`numerator()/denominator()`

as <u>the</u> salient attribute?

$$\frac{1}{\not{0}} == \frac{2}{\not{0}} ?$$

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

`numerator()/denominator()`

as <u>the</u> salient attribute?

$$\frac{1}{2} == \frac{-1}{-2} ?$$

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

`numerator()/denominator()`

as the salient attribute?

$$\frac{1}{2} == \frac{100}{200} ?$$

# Value-Semantic Properties

## Selecting Salient Attributes

**What about**

`numerator()/denominator()`

as <u>the</u> salient attribute?

$$\left\lfloor \frac{1}{2} \right\rceil^{10} == \left\lfloor \frac{100}{200} \right\rceil^{10} ?$$

# Value-Semantic Properties

## Selecting Salient Attributes

?

**What about**

~~`numerator()/denominator()`~~

as <u>the</u> salient attribute?

$$\left[\frac{1}{2}\right]_{10} == \left[\frac{100}{200}\right]_{10} \ ?$$

***VIOLATES SUBTLE <u>ESSENTIAL</u> PROPERTY OF VALUE***

# Value-Semantic Properties

## Selecting Salient Attributes

If you choose to make

`numerator()/denominator()`

a salient attribute

**(probably a bad idea)**

then do <u>not</u> expose numerator and denominator as separate attributes…

# Value-Semantic Properties

S... ...s

If ...

... ( )

a sa...

…or maintain them in "canonical form" (which may be computationally expensive).

(probably a bad idea)

then do <u>not</u> expose numerator and denominator as separate attributes…

# Value-Semantic Properties

## Selecting Salient Attributes

Guideline

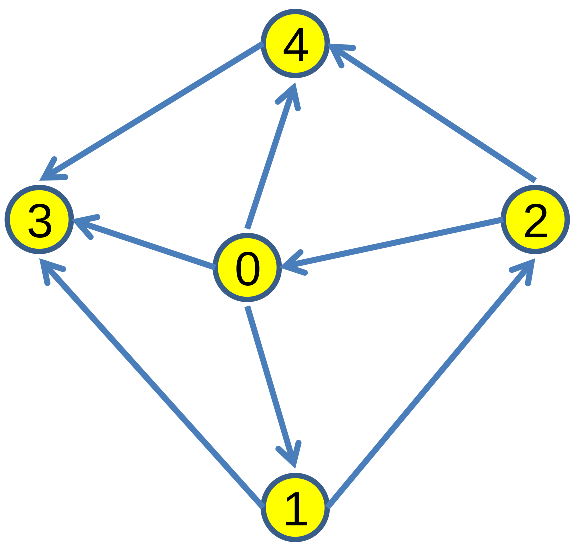
If two objects have *the same* value then the values of each ***observable attribute*** that ***contributes to value*** *should* respectively ***compare equal***.

# When should we omit valid value syntax?

# Value-Semantic Properties

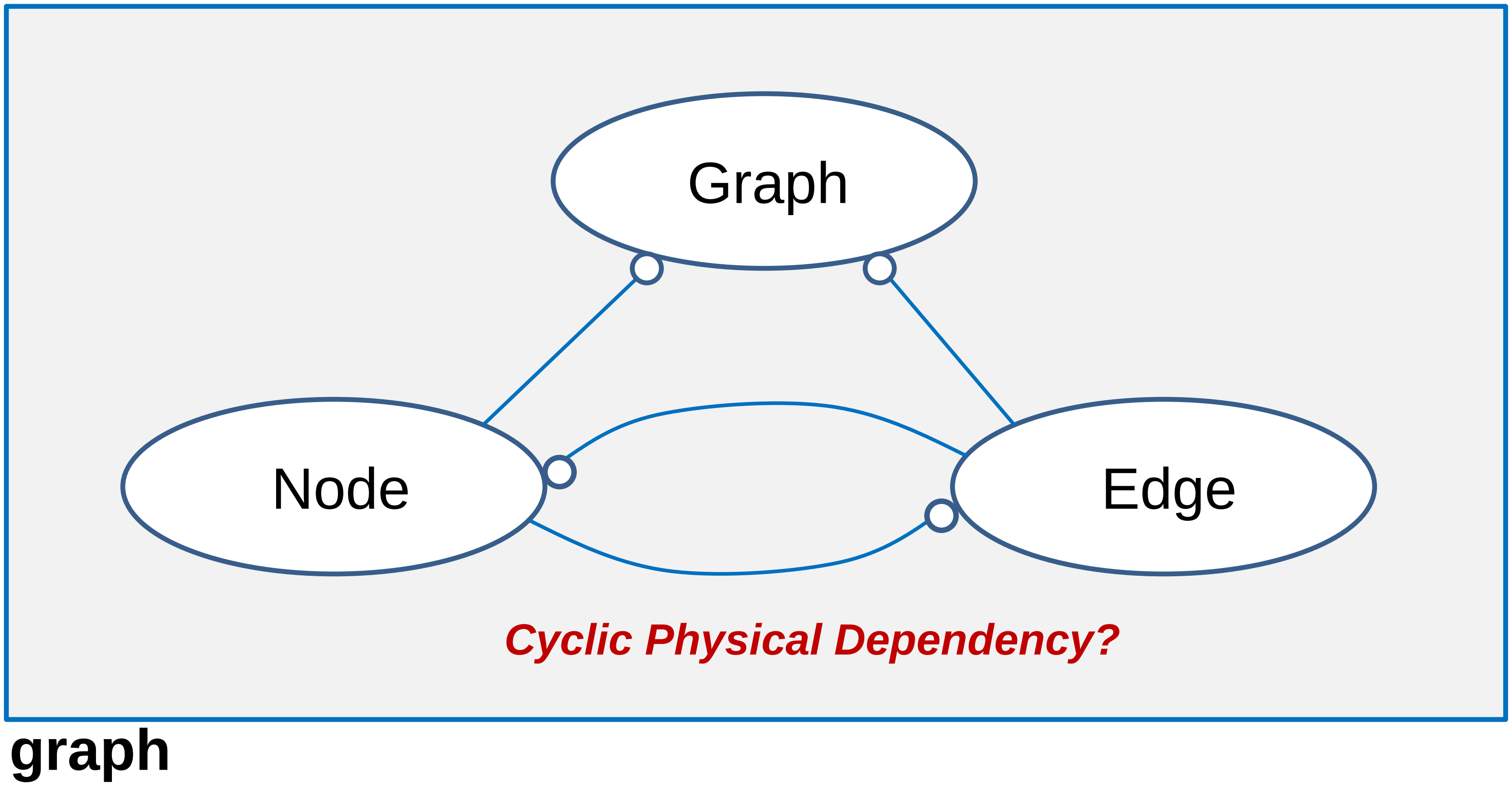
## Selecting Salient Attributes

# Graphs



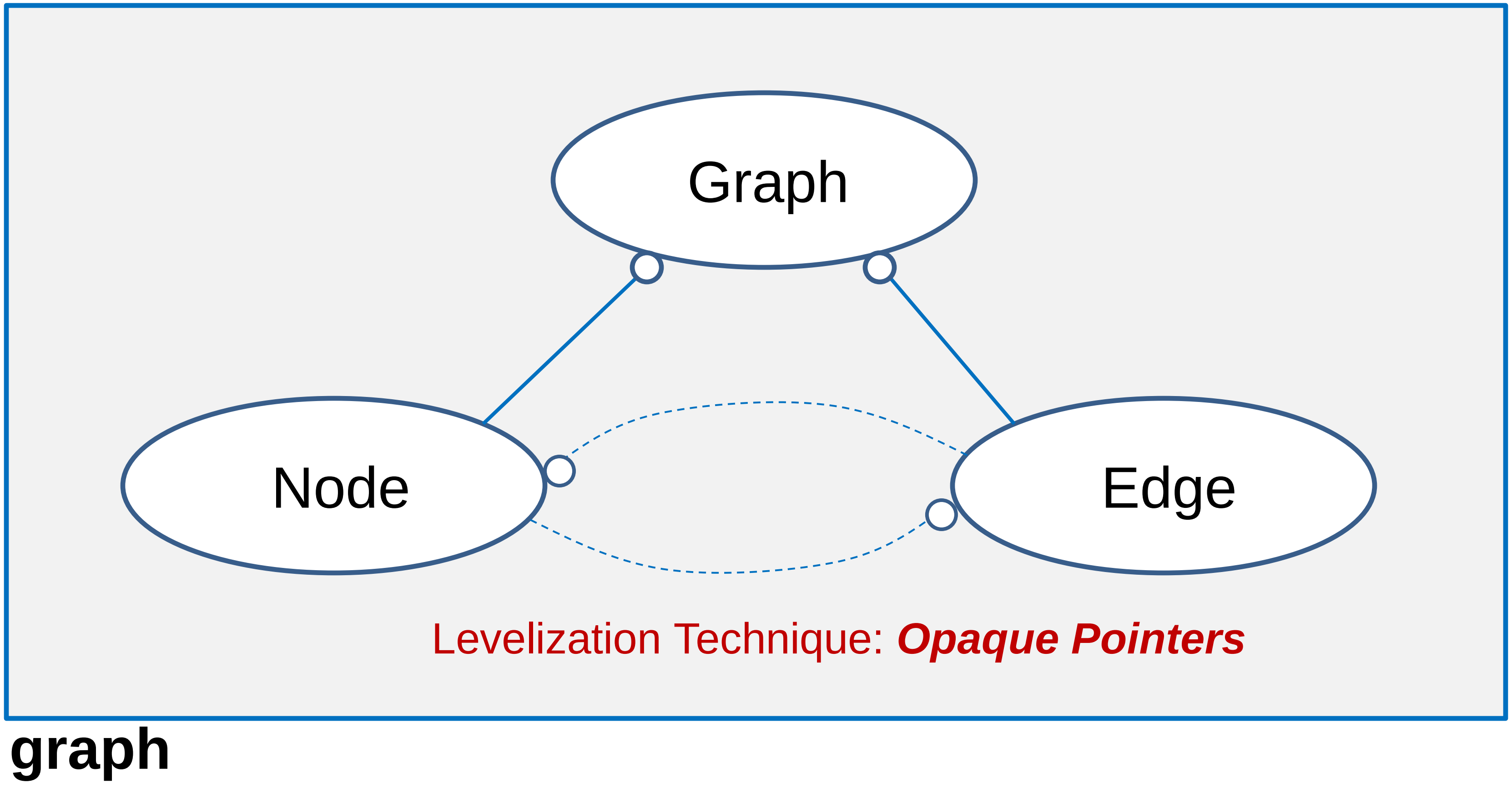
# Value-Semantic Properties

## Selecting Salient Attributes



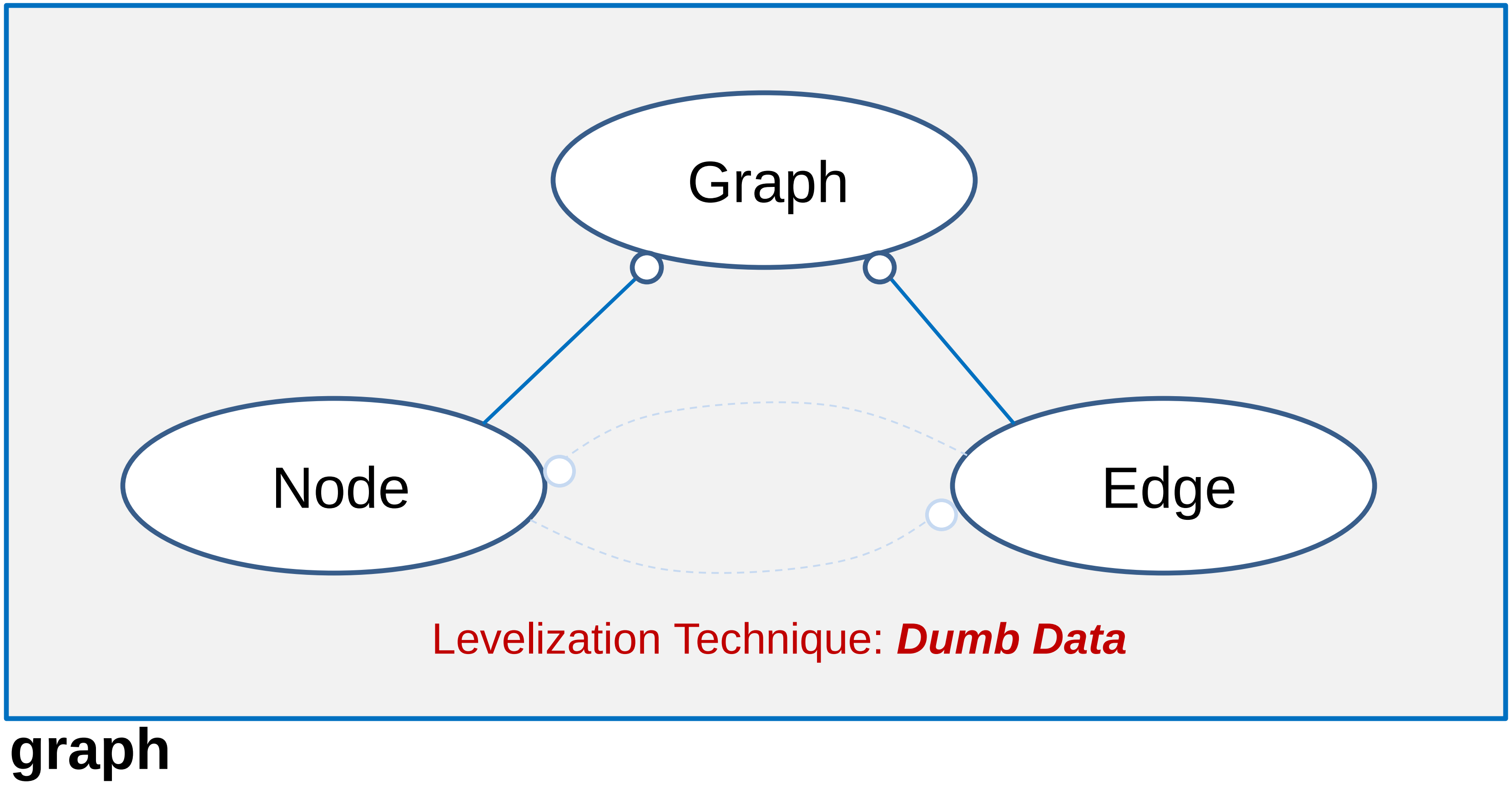
# Value-Semantic Properties

## Selecting Salient Attributes



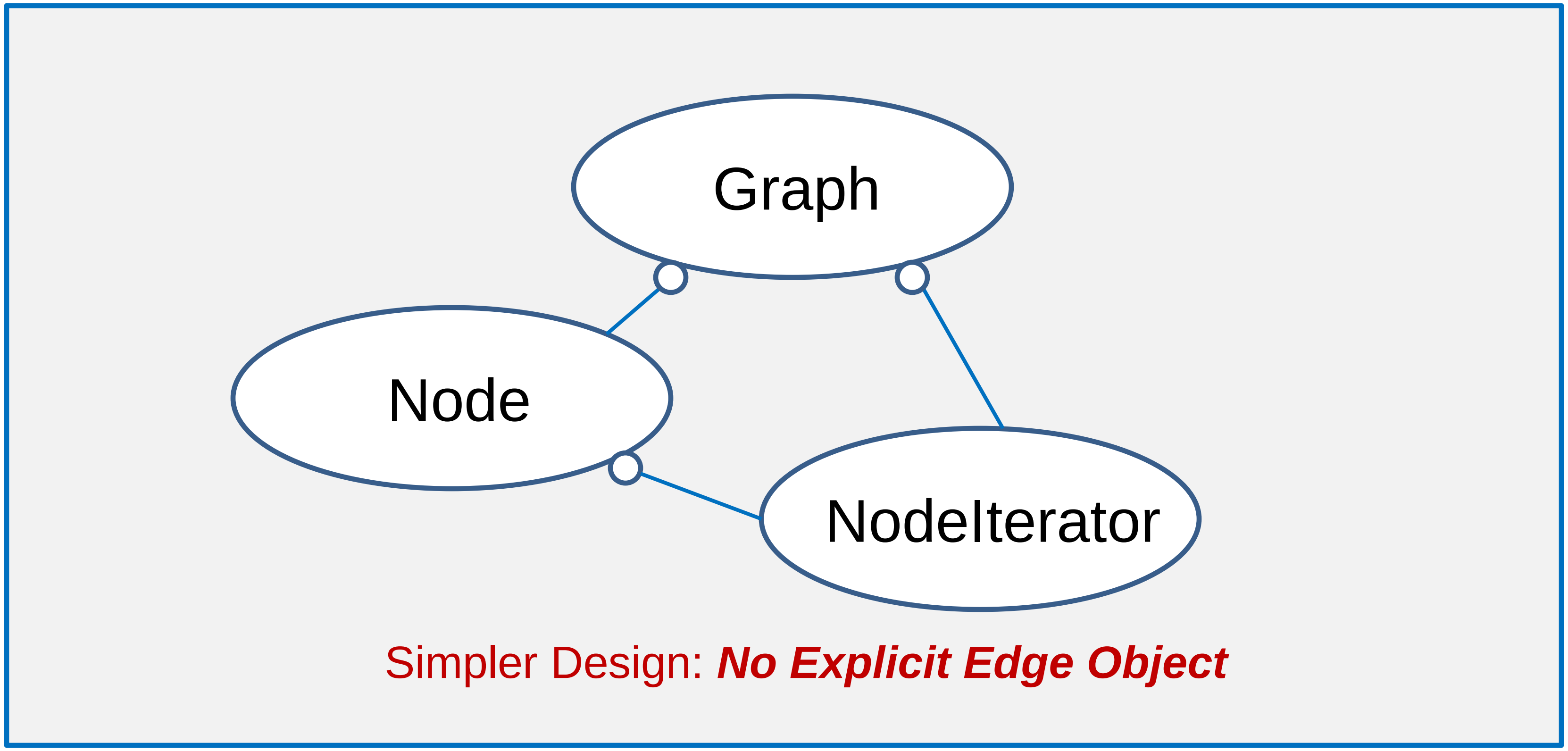
# Value-Semantic Properties

## Selecting Salient Attributes



# Value-Semantic Properties

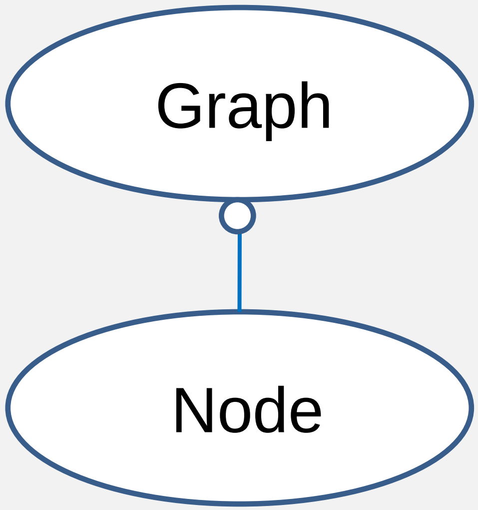
## Selecting Salient Attributes




**graph**

# Value-Semantic Properties

## Selecting Salient Attributes



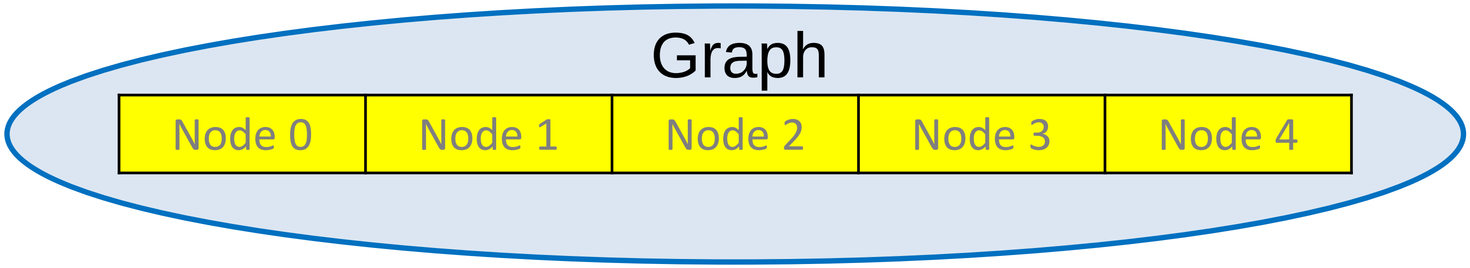

Yet Simpler Design: ***No Explicit NodeIterator***

**graph**

# Value-Semantic Properties

## Selecting Salient Attributes

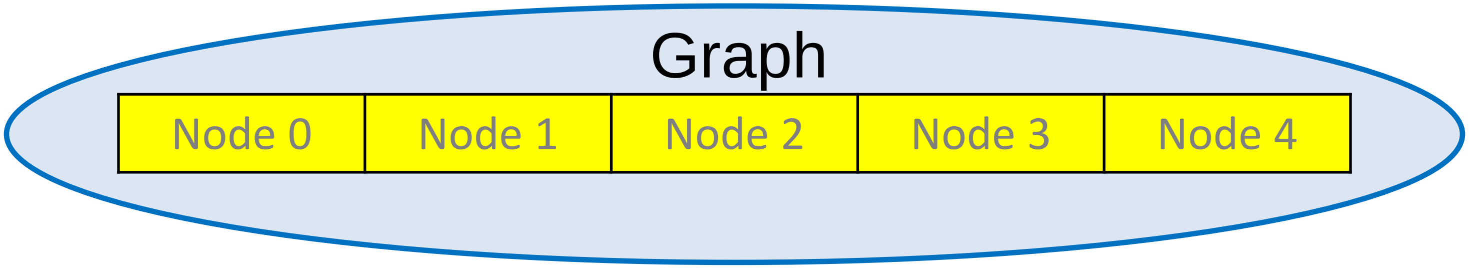
```
class Graph {
    // …
  public:
    // …
    int numNodes() const;
    const Node& node(int index) const;
};
// …
```



# Value-Semantic Properties

## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

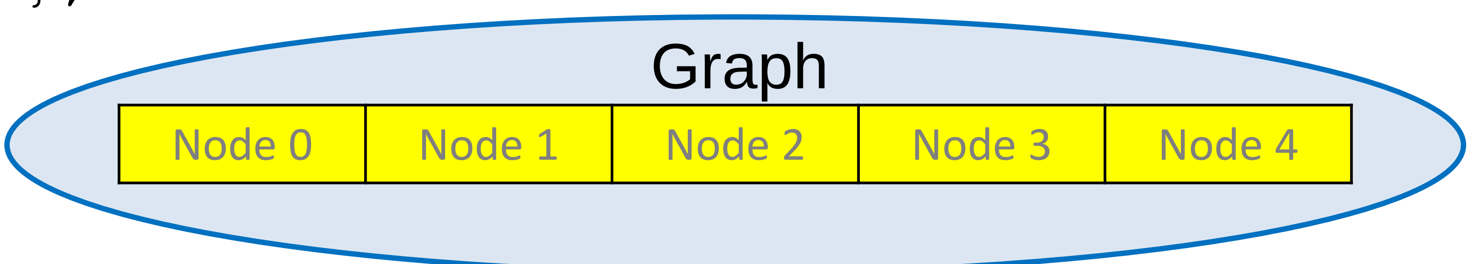


# Value-Semantic Properties

## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

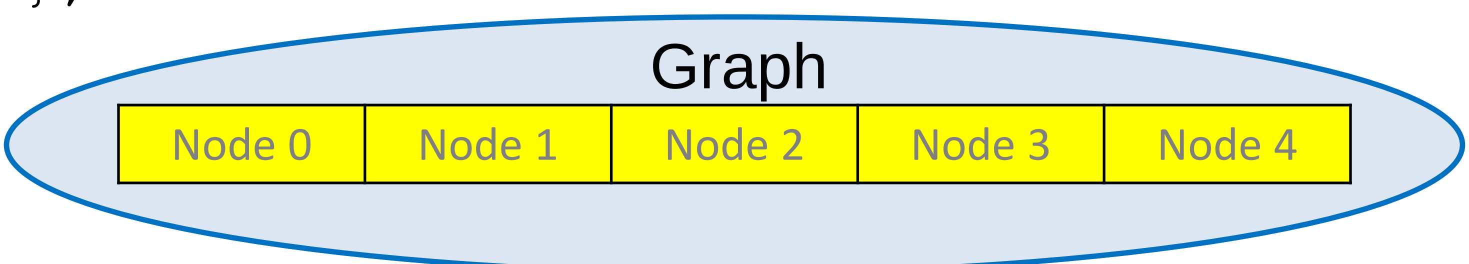
Really should be declared `const` but there's no room!



# Value-Semantic Properties

## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

# Value-Semantic Properties

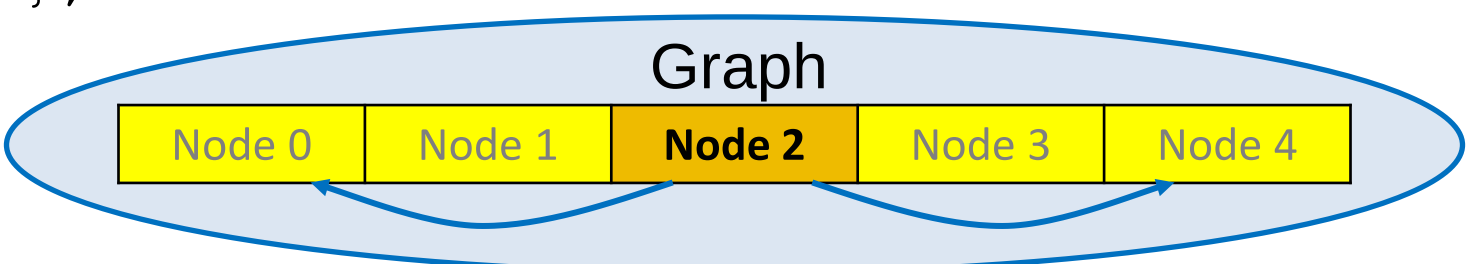
## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

2

2

Node 0
and
Node 4



# Value-Semantic Properties

## Selecting Salient Attributes

```
class Graph {
    // …
  public:
    // …
    int numNodes() const;
    const Node& node(int index) const;
};
// …
bool operator==(const Graph& lhs,
                const Graph& rhs);
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
class Graph {
    // …
  public:
    // …
    int numNodes() const;
    const Node& node(int index) const;
};
// …
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if …???
```

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.
- Number of edges.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.
- Number of edges.
- Number of nodes adjacent to each node.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.
- Number of edges.
- Number of nodes adjacent to each node.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.
- Number of edges.
- Number of nodes adjacent to each node.
- Specific nodes adjacent to each node.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

- Number of nodes.
- Number of edges.
- Number of nodes adjacent to each node.
- Specific nodes adjacent to each node.

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and,
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and, for each node index 'i'
    // '(0 <= i < N)',
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and, for each node index 'i'
    // '(0 <= i < N)', the nodes adjacent to
    // node 'i' in 'lhs' have the same
    // indices as those of the nodes
    // adjacent to node 'i' in 'rhs'.
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
class Node {
    // …
  public:
    // …
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
```

Maintained in *sorted* order?

Is "edge" *order* a *salient attribute*?

# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

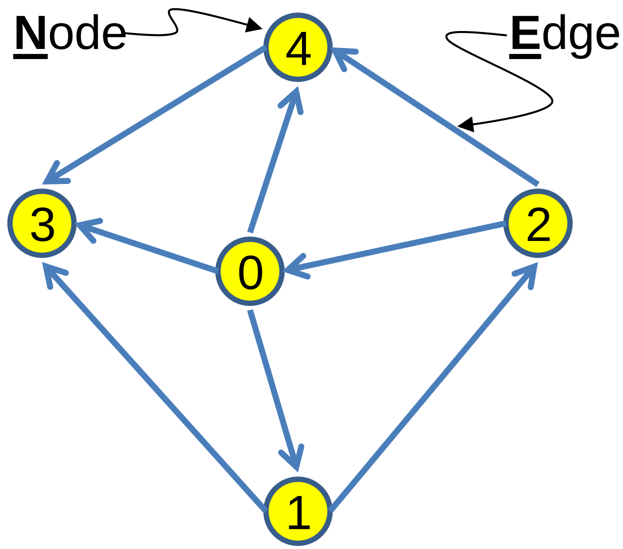
0: 4 1 3

1: 3 2

2: 0 4

3:

4: 3




**Ordered Edges**

0: 1 3 4

1: 2 3

2: 0 4

3:

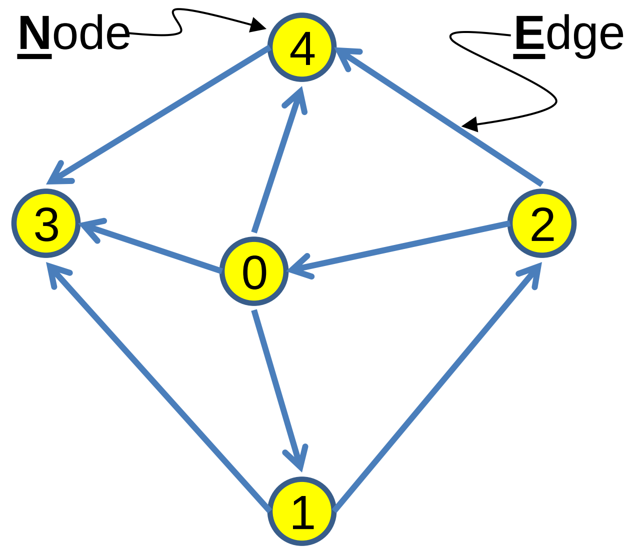
4: 3

# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

0: 4 1 3
1: 3 2
2: 0 4
3:
4: 3




?

**O[**`operator==`**]**

**Ordered Edges**

0: 1 3 4
1: 2 3
2: 0 4
3:
4: 3

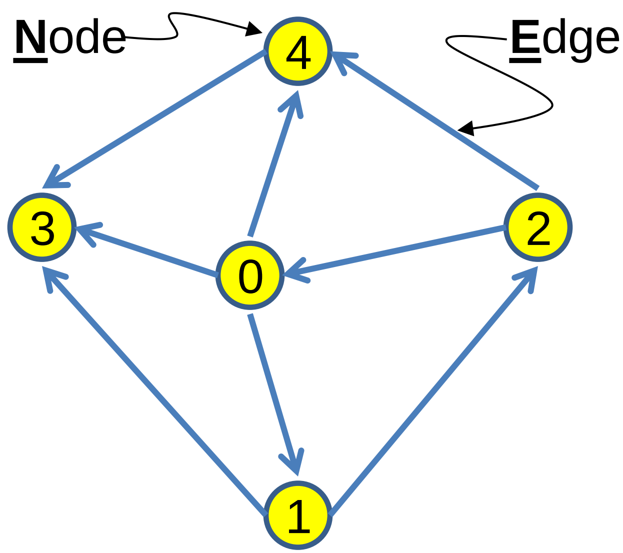
# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

0: 4 1 3
1: 3 2
2: 0 4
3:
4: 3

O[N + E²]




?

O[operator==]

**Ordered Edges**

0: 1 3 4
1: 2 3
2: 0 4
3:
4: 3

# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

0: 4 1 3

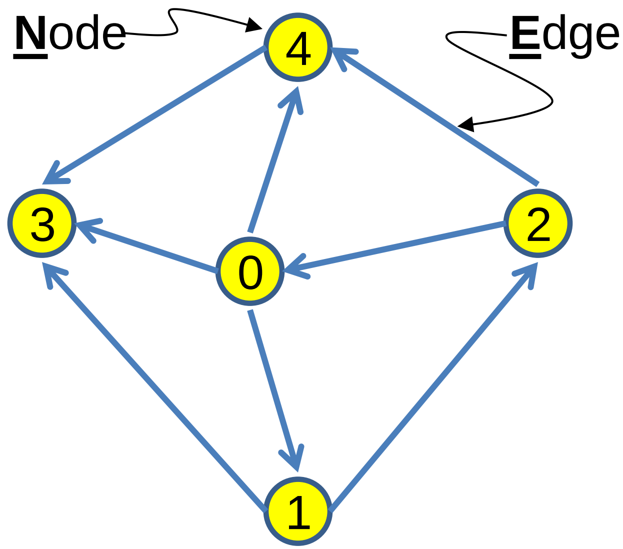
1: 3 2

2: 0 4

3:

4: 3

**O[N + E²]**




?

**O[**`operator==`**]**

**Ordered Edges**

0: 1 3 4

1: 2 3

2: 0 4

3:

4: 3

**O[N + E]**

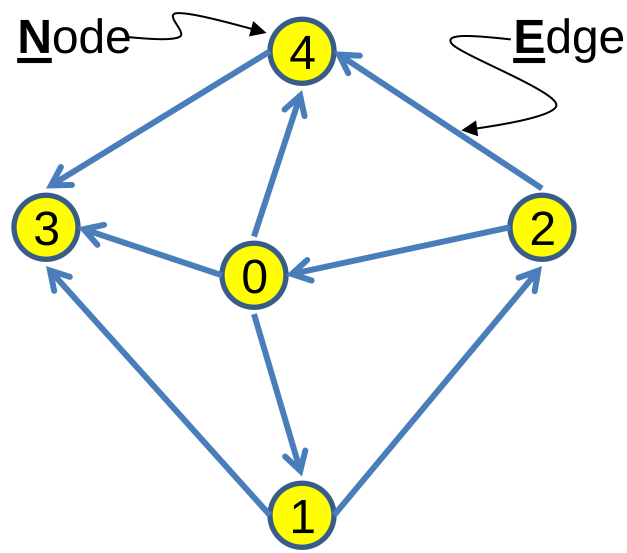
# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

0: 4 1 3
1: 3 2
2: 0 4
3:
4: 3

**O[N + E²]**




?

**O[**`operator==`**]**

**Ordered Edges**

0: 1 3 4
1: 2 3
2: 0 4
3:
4: 3

**O[N + E]**

Note that we *could* make it **O[N + E*Log(E)].**

# Value-Semantic Properties

## Selecting Salient Attributes

**Unordered Edges**

0: 4 1 3

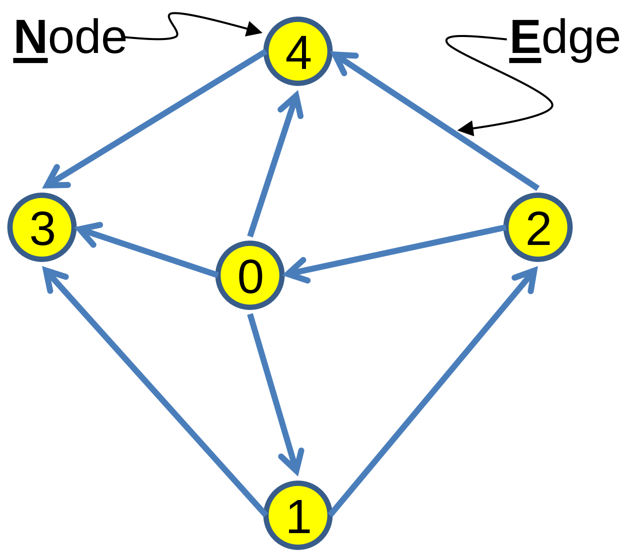
1: 3 2

2: 0 4

3:

4: 3

**O[N + E²]**




?

**O[**`operator==`**]**

**Ordered Edges**

0: 1 3 4

1: 2 3

2: 0 4

3:

4: 3

O[N + E]

Note that we *could* make it **O[N + E*Log(E)]**.

# Value-Semantic Properties

**OBSERVATION**

# Value-Semantic Properties

**OBSERVATION**

Value Syntax: **Not all or nothing!**

# Value-Semantic Properties

**OBSERVATION**

## Value Syntax: **Not all or nothing!**

An `std::set<int>` is a *value-semantic* type.

# Value-Semantic Properties

**OBSERVATION**

## Value Syntax: **Not all or nothing!**

An `std::set<int>` is a *value-semantic* type.

An `std::unordered_set<int>` is a *value-semantic* type,

# Value-Semantic Properties

## OBSERVATION

Value Syntax: **Not all or nothing!**

An `std::set<int>` is a *value-semantic* type.

An `std::unordered_set<int>` is a *value-semantic* type, **except that – until 2010 – it did not provide an** `operator==`.

# Value-Semantic Properties

## OBSERVATION

Value Syntax: **Not all or nothing!**

An `std::set<int>` is a *value-semantic* type.

An `std::unordered_set<int>` is a *value-semantic* type, **except that – until 2010 – it did not provide an** `operator==`.

**In large part due to performance concerns!**

# Value-Semantic Properties

**OBSERVATION**

Value Syntax: **Not all or nothing!**

A…

**NOT REGULAR!**

…`unordered_set<int>` is a *value-semantic* type, **except that – until 2010 – it did not provide an** `operator==`.

**In large part due to performance concerns!**

# Value-Semantic Properties

**OBSERVATION**

Val

An

**NOT CLEAR!**

**Excellent Starting Point!**

*value-*

*semantic*

**it did**

**not pro**

**In large p**

**ncerns!**

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓**Number of nodes.**

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓Number of nodes.

✓**Specific nodes adjacent to each node.**

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓Number of nodes.

✓Specific nodes adjacent to each node.

**X <u>Not</u> adjacent-node (i.e., edge) order.**

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓Number of nodes.

✓Specific nodes adjacent to each node.

X Not adjacent-node (i.e., edge) order.

➤ **What about node indices?**

(I.e., the *numbering* of the nodes)

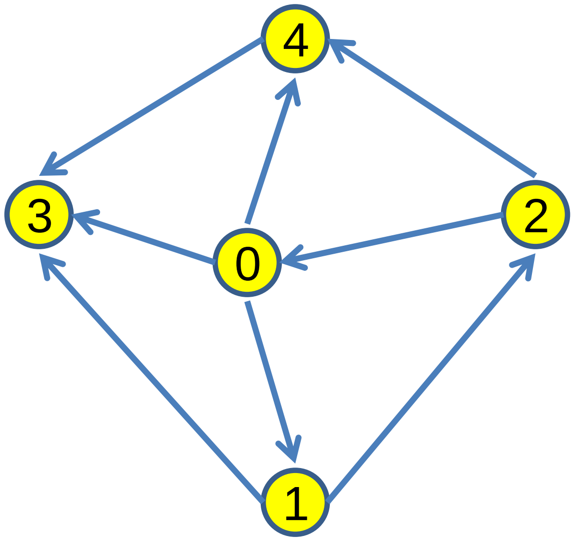
# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and there exists a renumbering
    // of the nodes in 'rhs' such that, for
    // each node-index 'i' '(0 <= i < N)',
    // the nodes adjacent to node 'i' in 'lhs'
    // have the same indices as those of the
    // nodes adjacent to node 'i' in 'rhs'.
```

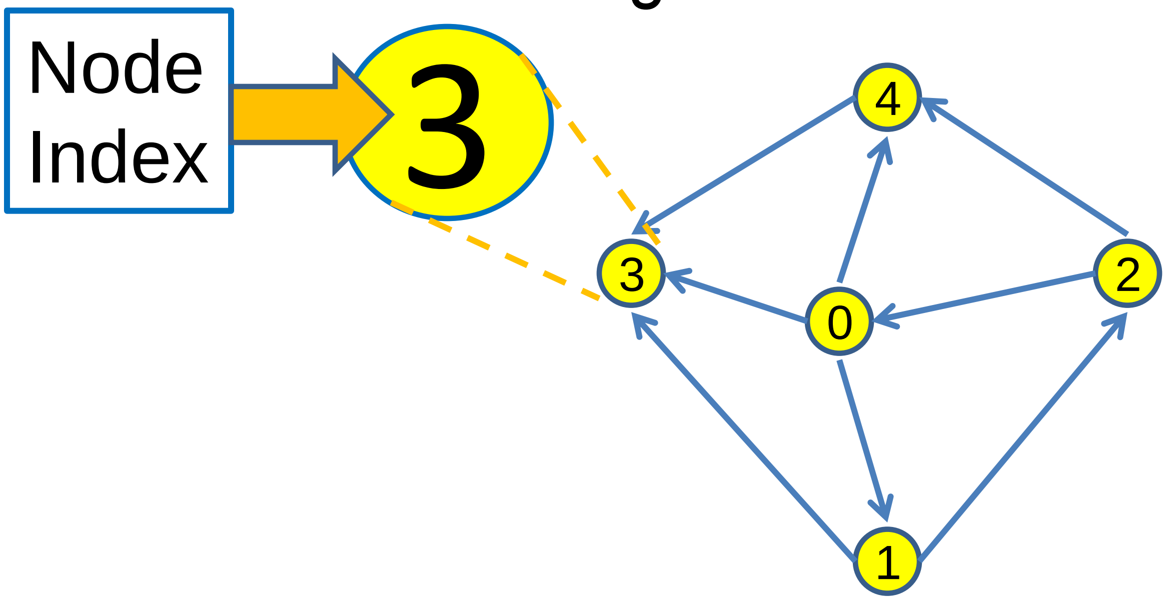
# Value-Semantic Properties

## Selecting Salient Attributes



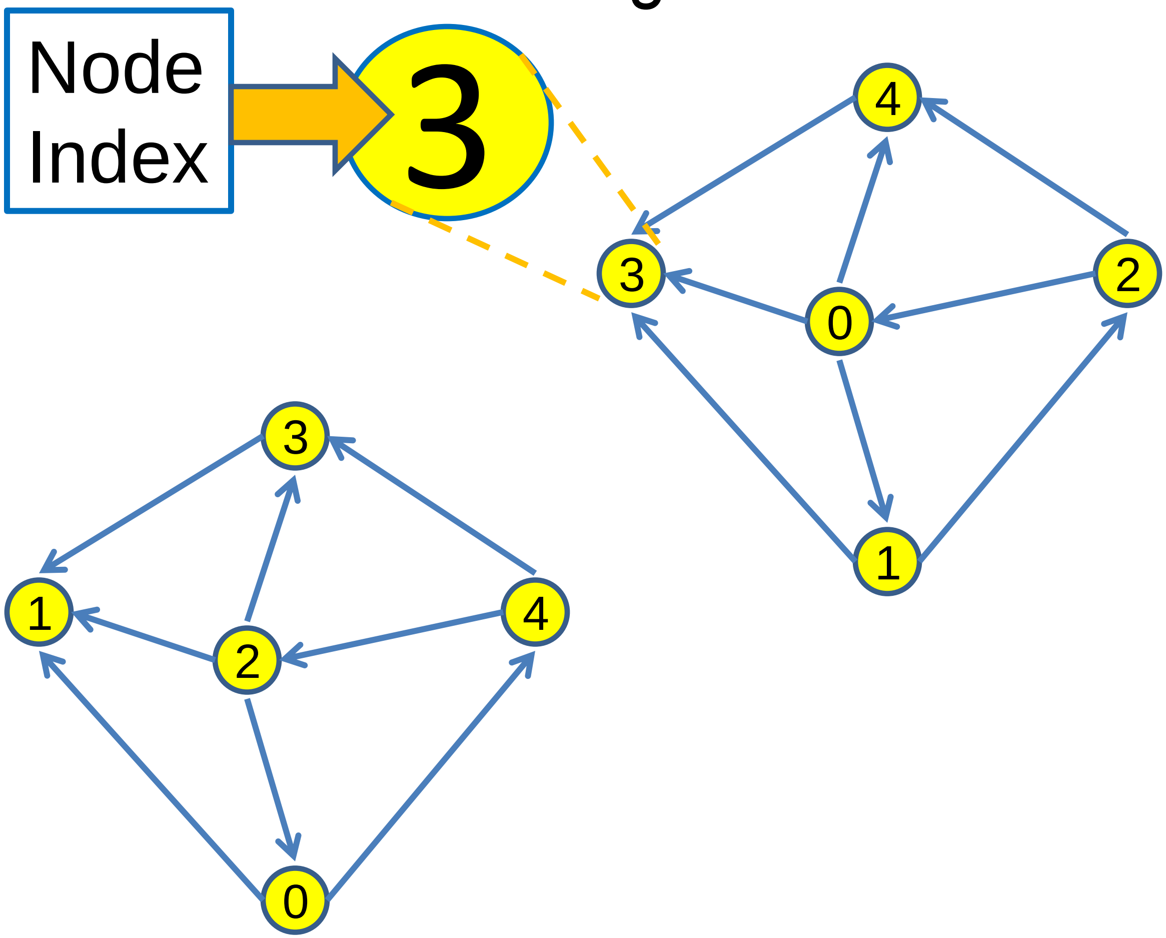
# Value-Semantic Properties

## Selecting Salient Attributes



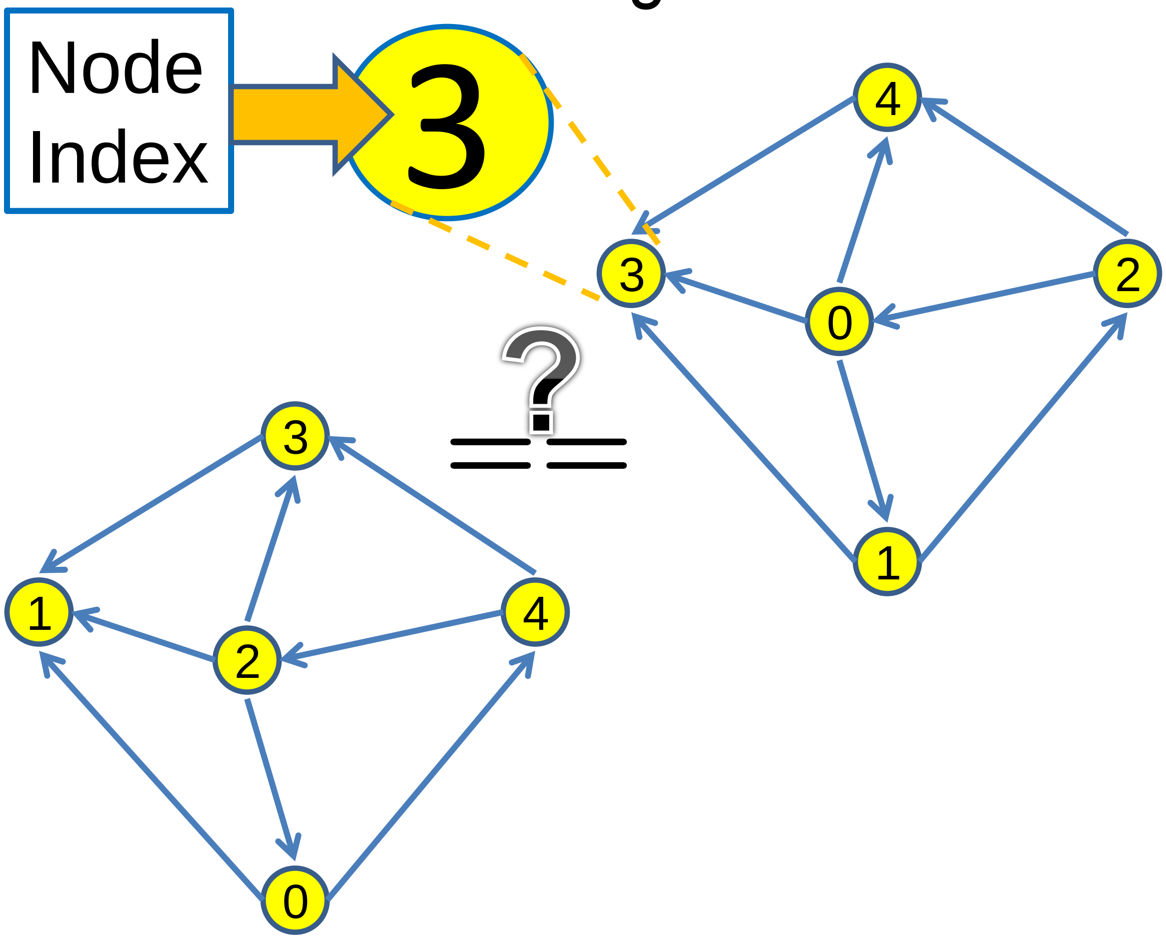
# Value-Semantic Properties

## Selecting Salient Attributes



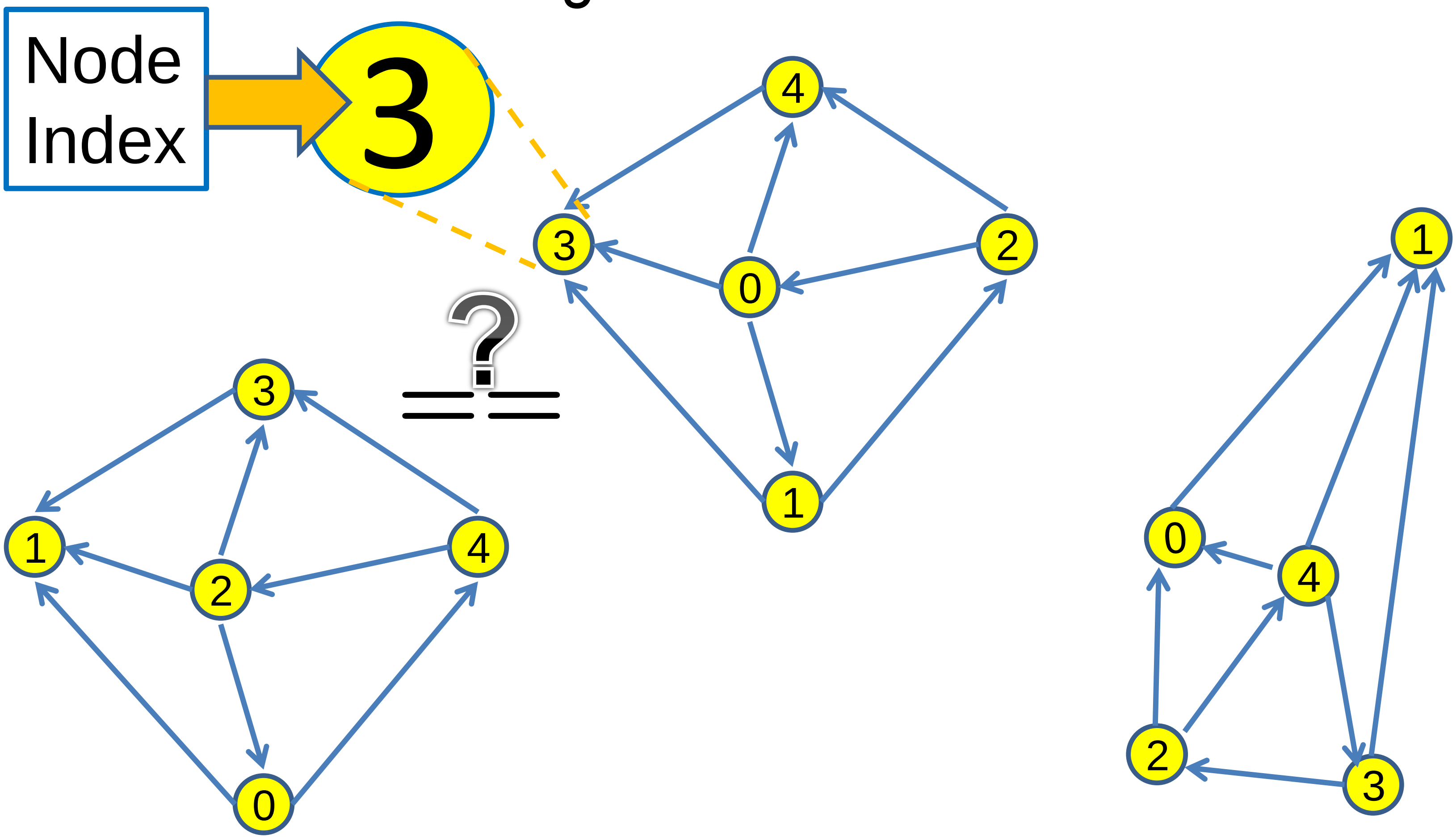
# Value-Semantic Properties

## Selecting Salient Attributes



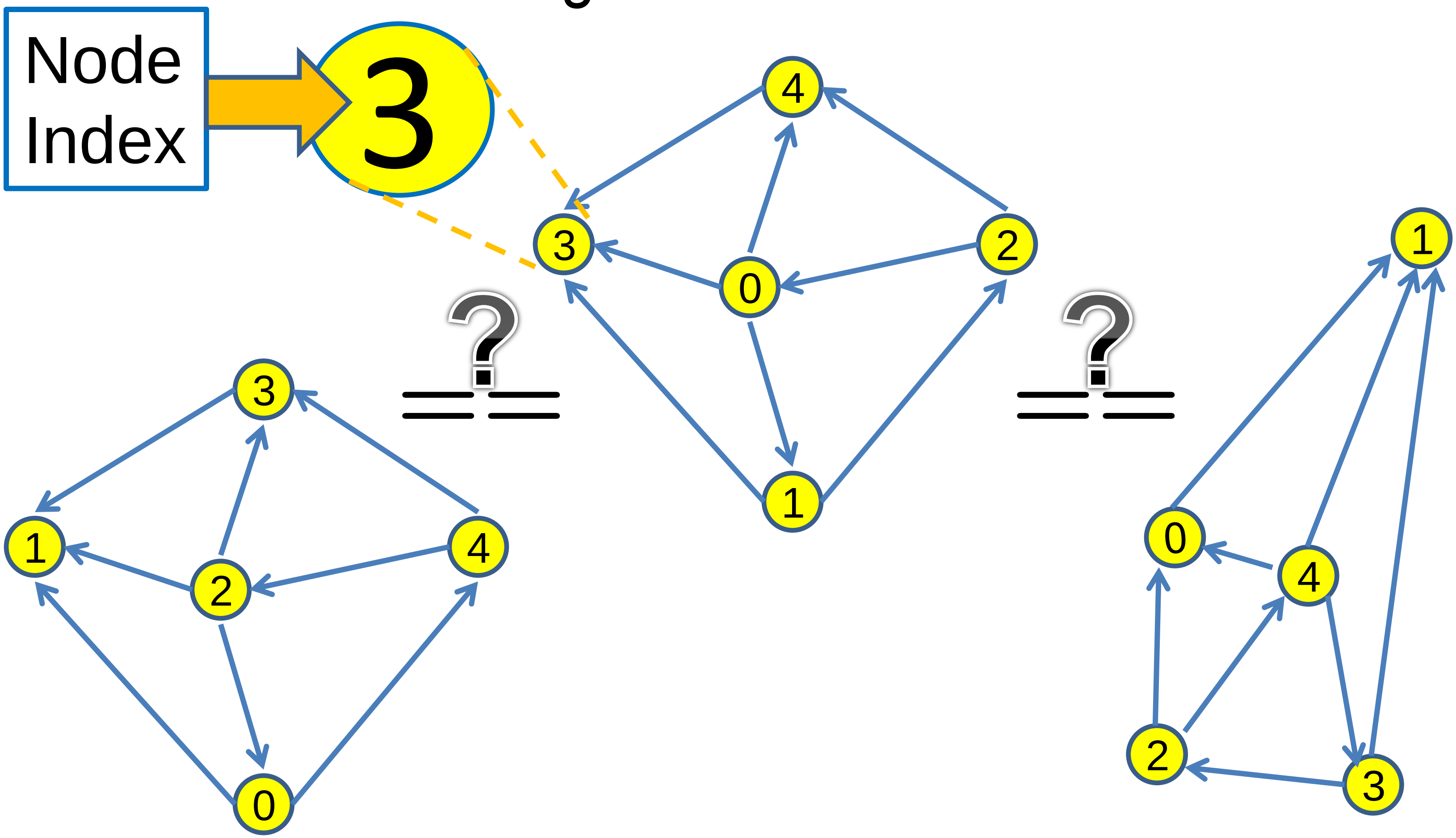
# Value-Semantic Properties

## Selecting Salient Attributes



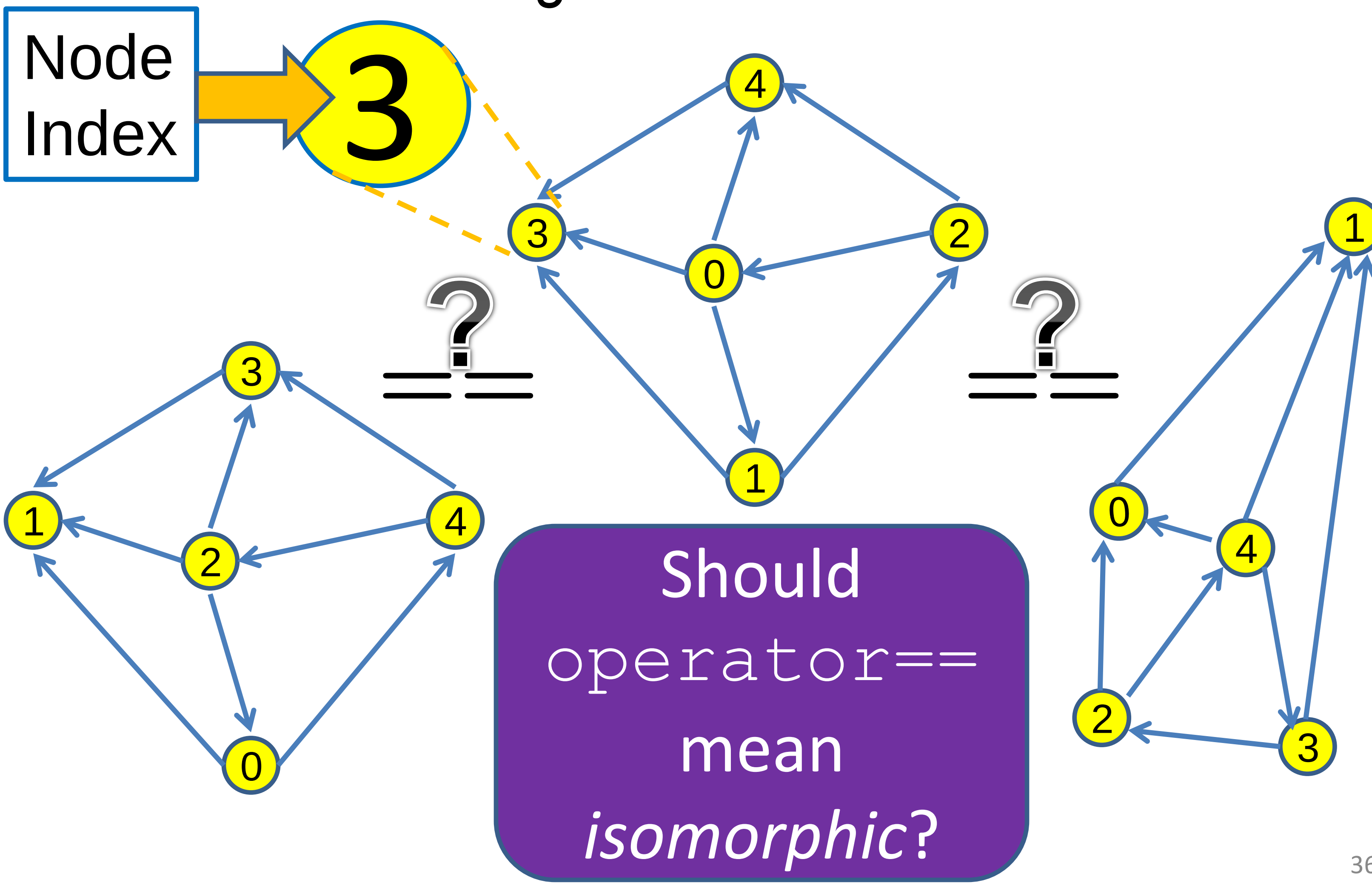
# Value-Semantic Properties

## Selecting Salient Attributes



# Value-Semantic Properties

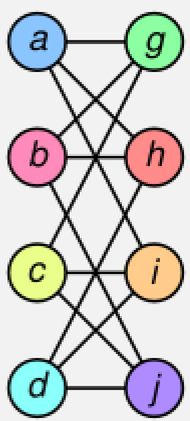
## Selecting Salient Attributes

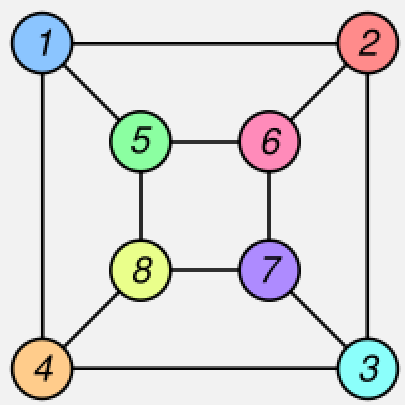


# Value-Semantic Properties

## Selecting Salient Attributes

In graph theory, an **isomorphism of graphs*** *G* and *H* is a bijection *f* between the vertex sets of *G* and *H* such that any two vertices *u* and *v* of *G* are adjacent in *G* if and only if *f*(*u*) and *f*(*v*) are adjacent in *H*.

<table>
<tr><th>Graph G</th><th>Graph H</th><th>An isomorphism between G and H</th></tr>
<tr><td>


</td><td>


</td><td>f(a) = 1<br>f(b) = 6<br>f(c) = 8<br>f(d) = 3<br>f(g) = 5<br>f(h) = 2<br>f(i) = 4<br>f(j) = 7</td></tr>
</table>

*http://en.wikipedia.org/wiki/Graph_isomorphism

# Value-Semantic Properties

## Selecting Salient Attributes

How hard is it to determine

*Graph Isomorphism*?

# Value-Semantic Properties

## Selecting Salient Attributes

How hard is it to determine

*Graph Isomorphism*?

**Is known to be in NP *and* CO-NP.**

# Value-Semantic Properties

## Selecting Salient Attributes

How hard is it to determine

*Graph Isomorphism*?

**Is known to be in NP *and* CO-NP.**

**Not known to be NP Complete.**

# Value-Semantic Properties

## Selecting Salient Attributes

How hard is it to determine

*Graph Isomorphism*?

**Is known to be in NP *and* CO-NP.**

**Not known to be NP Complete.**

**Not known to be in P (Polynomial time).**

# Value-Semantic Properties

## Selecting Salient Attributes

How hard is it to determine

*Graph Isomorphism*?

Is known to be in NP *and* CO-NP.

Not known to be NP Complete.

Not known to be in P (Polynomial time).

It's Hard!

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and there exists a renumbering
    // of the nodes in 'rhs' such that, for
    // each node-index 'i' '(0 <= i < N)',
    // the nodes adjacent to node 'i' in 'lhs'
    // have the same indices as those of the
    // nodes adjacent to node 'i' in 'rhs'.
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and there exists a renumbering
    // of the nodes in 'rhs' such that, for
    // each node-index 'i' '(0 <= i < N)',
    // the nodes adjacent to node 'i' in 'lhs'
    // have the same indices as those of the
    // nodes adjacent to node 'i' in 'rhs'.
```

# Value-Semantic Properties

## Selecting Salient Attributes

```
bool operator==(const Graph& lhs,
                const Graph& rhs);
    // Two 'Graph' objects have the same
    // value if they have the same number of
    // nodes 'N' and, for each node-index 'i'
    // '(0 <= i < N)', the ordered sequence
    // of nodes adjacent to node 'i' in
    // 'lhs' has the same value as the one
    // for node 'i' in 'rhs'.
```

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓Number of nodes.

✓Specific nodes adjacent to each node.

# Value-Semantic Properties

## Selecting Salient Attributes

What are the salient attributes of `Graph`?

✓Number of nodes.

✓Specific nodes adjacent to each node.

And, as a practical matter,

✓***Numbering* of the nodes.**

# Value-Semantic Properties

## Selecting Salient Attributes

**Or else we <u>Must</u> Omit Equality Comparison Operators for this Class!**

✓ ***Numbering* of the nodes.**

# Value-Semantic Properties

## Selecting Salient Attributes

**Or else we <u>Must</u> Omit Equality Comparison Operators for this Class!**

✓ ***Numbering* of the nodes.**

**AND PERHAPS PROVIDE THIS FUNCTIONALITY IN A UTILITY**

# Discussion

Why would we ever omit valid value syntax when there is only one obvious notion of value?

# Discussion

Why would we ever omit valid value syntax when there

**When We Cannot Do It Efficiently!**

# Discussion

Why would we ever omit valid value syntax when there

**When Doing So Is "Off Message!"**

# Value-Semantic Properties

## Selecting Salient Attributes

# (Summary So Far)

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

When selecting *salient* attributes, avoid subjective (domain-specific) interpretation:

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

When selecting *salient* attributes, avoid subjective (domain-specific) interpretation:

- Fractions may be ***equivalent***, but not ***the same***.

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

When selecting *salient* attributes, avoid subjective (domain-specific) interpretation:

- Fractions may be ***equivalent***, but not ***the same***.
- Graphs may be ***isomorphic***, yet ***distinct***.

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

When selecting *salient* attributes, avoid subjective (domain-specific) interpretation:

- Fractions may be ***equivalent***, but not ***the same***.
- Graphs may be ***isomorphic***, yet ***distinct***.
- Triangles may be ***similar*** and still ***differ***.

**DON'T "EDITORIALIZE" EQUALITY**

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any “subjective interpretations” of *equality* to ***named functions!***

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any "subjective interpretations" of *equality* to ***named functions*** – ideally, in *higher-level* components:

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any "subjective interpretations" of *equality* to ***named functions*** – ideally, in *higher-level* components:

```
struct MyUtil {
    static bool areEquivalent(const Rational& a)
                              const Rational& b);
    static bool areIsomorphic(const Graph& g1,
                              const Graph& g2);
    static bool areSimilar(const Triangle& x,
                           const Triangle& y);
};
```

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any "subjective interpretations" of *equality* to ***named functions*** – ideally, in *higher-level* components:

```
struct MyUtil {
    static bool areEquivalent(const Rational& a)
                              const Rational& b);
    static bool areIsomorphic(const Graph& g1,
                              const Graph& g2);
    static bool areSimilar(const Triangle& x,
                           const Triangle& y);
};
```

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any "subjective interpretations" of *equality* to ***named functions*** – ideally, in *higher-level* components:

```
struct MyUtil {
    static bool areEquivalent(const Rational& a)
                              const Rational& b);
    static bool areIsomorphic(const Graph& g1,
                              const Graph& g2);
    static bool areSimilar(const Triangle& x,
                           const Triangle& y);
};
```

# Value-Semantic Properties

## Selecting Salient Attributes

(Summary So Far)

Relegate any "subjective interpretations" of *equality* to ***named functions*** – ideally, in *higher-level* components:

```
struct MyUtil {
    static bool areEquivalent(const Rational& a)
                              const Rational& b);
    static bool areIsomorphic(const Graph&
                              cons
    static bool areSimil                    x,
                                    angle& y);
};
```

**Utility Class Category**

# Value-Semantic Properties

A collateral benefit is Terminology:

**Saying what we mean facilitates understanding.**

# Collateral Benefit: Terminology

"…objects are the same…"

"…objects are identical…"

"…objects are equal…"

"…objects are equivalent…"

"…create a copy of…"

# Collateral Benefit: Terminology

"...objects are the same..."

"...objects are identical..."

"...objects are equal..."

"...objects are equivalent..."

"...create a copy of..."

# Collateral Benefit: Terminology

**BE PRECISE!**

"...objects are the same..."

"...objects are identical..."

"...objects are equal..."

"...objects are equivalent..."

"...create a copy of..."

# Collateral Benefit: Terminology

## BE PRECISE!

"…objects are the same…"

"…objects are identical…"

"…objects are equal…"

"…objects are equivalent…"

"…create a copy of…"

**SAY EXACTLY *WHAT* MUST BE THE SAME!**

# Collateral Benefit: Terminology

"…objects are the same…"

"…objects are identical…"

(identity)

# Collateral Benefit: Terminology

"…objects are the same…"

"…objects are identical…"

(identity)

**"…**(aliases) **refer to the same object…"**

# Collateral Benefit: Terminology

"…objects are the same…"

(value)

# Collateral Benefit: Terminology

"…objects are the same…"

(value)

**"…**(objects) **have the same value…"**

# Collateral Benefit: Terminology

"...objects are the same..."

(value)

"...(objects) **have the same value**..."

"...(objects) **refer to the same value**..."

# Collateral Benefit: Terminology

"…objects are the same…"

(value)

"…(objects) **have the same value…**"

"…(objects) **refer to the same value…**"

"…(objects) **represent the same value…**"

# Collateral Benefit: Terminology

"…objects are the same…"

"…objects are equal…"

(equality)

# Collateral Benefit: Terminology

"…objects are the same…"

"…objects are equal…"

(equality)

**"…**(objects) **compare equal…"**

# Collateral Benefit: Terminology

"...objects are the same..."

"...objects are equal..."

(equality)

**"...**(objects) **compare equal...**"

**"...**(homogeneous) `operator==` **returns** `true`**...**"

# Collateral Benefit: Terminology

"...objects are the same..."

"...objects are equal..."

(equality)

**"...**(objects) **compare equal...**"

"...(homogeneous) `operator==` returns `true`..."

***For value-semantic objects:***

# Collateral Benefit: Terminology

"...objects are the same..."

"...objects are equal..."

(equality)

**"...**(objects) **compare equal..."**

**"...**(homogeneous) `operator==` **returns** `true`**..."**

**For *value-semantic* objects:**

**Means have the *same value!***

# Collateral Benefit: Terminology

"…objects are the same…"
(equivalent)

# Collateral Benefit: Terminology

"…objects are the same…"

(equivalent)

**In separate *named* functions:**

# Collateral Benefit: Terminology

"…objects are the same…"

(equivalent)

**In separate *named* functions:**

"…fractions are equivalent…"

# Collateral Benefit: Terminology

"...objects are the same..."

(equivalent)

**In separate *named* functions:**

"...fractions are equivalent..."

"...graphs are isomorphic..."

# Collateral Benefit: Terminology

"…objects are the same…"

(equivalent)

**In separate *named* functions:**

"…fractions are equivalent…"

"…graphs are isomorphic…"

"…triangles are similar…"

# Outline

1. Introduction and Background

   Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

   Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

   Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

   What must be remembered when designing value types

# Outline

1. Introduction and Background

    Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

    Most importantly, the ***Essential Property of Value***

3. Two Important, Instructional Case Studies

    Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

    What must be remembered when designing value types

# Regular Expressions

Important Design Questions:

- What is a *Regular Expression*?
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Regular Expressions

Important Design Questions:

- **What is a *Regular Expression*?**
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Regular Expressions

## What is a *Regular Expression*?

# Regular Expressions

## **What is a *Regular Expression*?**

A *Regular Expression* describes a language that can be *accepted* by a Finite-State Machine (FSM).

# Regular Expressions

## What is a *Regular Expression*?

A *Regular Expression* describes a language that can be *accepted* by a Finite-State Machine (FSM).

E.g.,

**(1|0)+** describes binary numbers.

# Regular Expressions

## Important Design Questions:

- What is a *Regular Expression*?
- **Why create a separate class for it?**
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Regular Expressions

## **Why create a separate class for it?**

# Regular Expressions

## **Why create a separate class for it?**

A *Regular-Expression* class imbued with the value of a regular expression can be used to determine whether (or not) arbitrary string tokens are members of the language that the regular-expression value denotes.

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language; accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

# Regular Expressions

## Why create a separate class for it?

Class Methods

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

# Regular Expressions

## Why create a separate class for it?

Creators

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

# Regular Expressions

## Why create a separate class for it?

Creators

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();   // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

Creators

# Regular Expressions

## Why create a separate class for it?

Creators

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

Manipulators

Whatever the value is.

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

Manipulators

What is the value?

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

Manipulators

Why both?

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language: Accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

Accessors

a.k.a.
"accept"
or
"matching"

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language; accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

**Which Operations Are Salient?**

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language; accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

**Which Operations Are *Salient?***

Just one!

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();   // Empty language; accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

**Which Operations Are *Salient*?**

Just one!

# Regular Expressions

## Why create a separate class for it?

```
class RegEx {
  // …
 public:
  static bool isValid(const char *regEx);
  RegEx();  // Empty language; accepts nothing.
  RegEx(const char *regEx);
  RegEx(const RegEx& other);
  ~RegEx();
  RegEx& operator=(const RegEx& rhs);
  void setValue(const char *regEx);
  int setValueIfValid(const char *regEx);
  bool isMember(const char *token) const;
};
```

**Which Operations Are *Salient*?**

Let's think about this!

# Regular Expressions

Important Design Questions:

- What is a *Regular Expression*?
- Why create a separate class for it?
- **Does/should it represent a value?**
- How should its value be defined?
- Should such a class be *regular*?

# Regular Expressions

# **Does/should it represent a value?**

# Regular Expressions

**Does/should it represent a value?**

Is a `RegEx` class a *value type*, or a *mechanism?*

# Regular Expressions

## **Does/should it represent a value?**

Is a `RegEx` class a *value type*, or a *mechanism?*

I.e., is there an obvious notion of what it means for two `RegEx` objects to have the same value?

# Regular Expressions

## **Does/should it represent a value?**

**I claim, "yes!"**

I.e., is there an obvious notion of what it means for two `RegEx` objects to have the same value?

# Regular Expressions

Important Design Questions:

- What is a *Regular Expression*?
- Why create a separate class for it?
- Does/should it represent a value?
- **How should its value be defined?**
- Should such a class be *regular*?

# Regular Expressions

## How should its value be defined?

1. **The string used to create it.**

# Regular Expressions

## How should its value be defined?

1. The string used to create it.
2. **The language it accepts.**

# Regular Expressions

**How should its value be defined?**

1. The string used to create it.
2. The language it accepts.

Note that there is no accessor to get the string used to initialize the value.

# Regular Expressions

**How should its value be defined?**

1. The string used to create it.
2. **The language it accepts.**

**IMO, the correct answer is 2. Why?**

Note that there is no accessor to get the string used to initialize the value.

# Regular Expressions

## How should its value be defined?

**Actually, there is no such accessor, precisely because we defined value the way we did!**

# Regular Expressions

## How should its value be defined?

1. Th

2

lue.

What makes a `RegEx` value special – i.e., distinct from that of the `(const char *)` used to create it – is the ***language value*** a `RegEx` object represents.

# Regular Expressions

## How should its value be defined?

1.

2.

**Th**

Be … et

th … e.

**Had we provided such an accessor, it would <u>not</u> be considered *salient*.**

# Regular Expressions

Just like *capacity* for

`std::vector`

**How … d?**

1.

2.

**Th**

Be … et

th … e.

**Had we provided such an accessor, it would <u>not</u> be considered *salient*.**

# Regular Expressions

Or *iteration order* for `std::unorderd_map`

1.

2.

**Th**

Be … et

th … .

**PERHAPS “FINITE AUTOMATA” WOULD HAVE BEEN A BETTER NAME?**

# Regular Expressions

Or *iteration order* for

`std::unorderd_map`

1.

2.

PERHAPS

"FINITE AUTOMATA"

Th

Be t

th

"Language"??

# Regular Expressions

Important Design Questions:

- What is a *Regular Expression*?
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- **Should such a class be *regular*?**

# Regular Expressions

## Should such a class be regular?

I.e., Should our `RegEx` class support all of the value-semantic syntax of a *regular* class?

# Regular Expressions

## **Should such a class be regular?**

I.e., Should our `RegEx` class support all of the value-semantic syntax of a *regular* class?

Question: How *expensive* would `operator==` be to implement?

# Regular Expressions

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

# Regular Expressions

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

In honor of this
very important question
would everyone
PLEASE STAND UP NOW!

# Regular Expressions

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

## Regular Expressions

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[1]?

# Regular Expressions

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

- O[1]?

**PLEASE SIT DOWN as soon as I have gone TOO FAR!**

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]

**I.e., Please sit down NOW if you can write `operator==` for `class RegEx` in O[1]!**

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]

## Regular Expressions

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]

# Regular Expressions

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]

**Please sit down NOW if you can write `operator==` for `class RegEx` in O[N]!**

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]

## Regular Expressions

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP
- NP complete

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP
- NP Complete
- P-SPACE

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP
- NP Complete
- P-SPACE
- P-SPACE Complete

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP
- NP Complete
- P-SPACE
- P-SPACE Complete
- Undecidable

3. Two Important, Instructional Case Studies

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

- O[log N]
- O[sqrt N]
- O[N]
- O[N * log N]
- O[N * sqrt N]
- O[N^2]
- O[N^2 * log N]
- Polynomial
- NP
- NP Complete
- P-SPACE
- **P-SPACE Complete**
- Undecidable

If you just sat down you were right!

# Regular Expressions

# P-Space Complete

Over an alphabet Σ, given one DFA having states $S=\{si\}$ (of which $A \subseteq S$ are accepting) and transition function $\delta: S \times \Sigma \rightarrow S$, *and* another DFA having states $T=\{tj\}$ (of which $B \subseteq T$ are accepting) and transition function $\zeta: T \times \Sigma \rightarrow T$, one can "easily" construct a DFA with states $U=S \times T$ (Cartesian product) and transition function $\eta((si, tj), \sigma) = (\delta(si, \sigma), \zeta(tj, \sigma))$, where $\sigma \in \Sigma$. Then the two original DFAs are equivalent iff the only states reachable in this Cartesian-product DFA are a subset of $(A \times B) \cup ((S \setminus A) \times (T \setminus B))$ — i.e., it's impossible to reach a state that is accepting in one of the original DFAs, but not in the other. Once one has translated the regular expressions to DFAs, the naive **time complexity is O[$|\Sigma|^{|S| \cdot |T|}$]**, and the **space complexity is O[$|S| \cdot |T| \cdot |\Sigma|$]**.

# Regular Expressions

## Should such a

Question: How *expensive* would

- O[log N]
- O[sqrt N]
-
-
-
-
-
-
-
-
- P-SPACE
- **P-SPACE Complete**
- Undecidable

Should we avoid value types where equality comparison is expensive?

**Clearly No Equality-Comparison Operators!**

# Regular Expressions

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

**Copy Construction and Assignment Aren't a Problem.**

- P-SPACE
- **P-SPACE Complete**
- Undecidable

## Regular Expressions

# Discussion?

# Priority Queues

Important Design Questions:

- What is a *Priority Queue*?
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Priority Queues

Important Design Questions:

- ***What is a Priority Queue?***
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Priority Queues

## **What is a *Priority Queue?***

# What is a *Priority Queue?*

A *priority queue* is a (generic) container that provides constant-time access to its **`top`** priority element – defined by a user-supplied *priority* function (or *functor*) – as well as supporting logarithmic-time **`push`**es and **`pop`**s of queue-element values.

# Priority Queues

## What is a *Priority Queue?*

Salient Operations

A *p* (generic) container that p e access to its `top` prio ent – defined by a user-supplied *priority* function (or *functor*) – as well as supporting logarithmic-time `push`es and `pop`s of queue-element values.

# What is a *Priority Queue?*

**Example Queue Element:**

```
class LabledPoint {
  std::string d_label;
  int         d_x;
  int         d_y;
 public:

  // … (Regular Type)

  const std::string& label() const { return d_label };
  int                x()     const { return d_x; };
  int                y()     const { return d_y; };
};

bool operator==(const LabeledPoint& lhs,
                const LabeledPoint& rhs) {
  return lhs.label() == rhs.label()
      && lhs.x()     == rhs.x()
      && lhs.y()     == rhs.y();        (Unconstrained Attribute Class)
}
```

# What is a *Priority Queue?*

**Example Queue Element:**

```
class LabledPoint {
  std::string d_label;
  int         d_x;
  int         d_y;
 public:

  // … (Regular Type)

  const std::string& label() const { return d_label };
  int                x()     const { return d_x; };
  int                y()     const { return d_y; };
};

bool operator==(const LabeledPoint& lhs,
                const LabeledPoint& rhs) {
  return lhs.label() == rhs.label()
      && lhs.x()     == rhs.x()
      && lhs.y()     == rhs.y();
}
```

**(Unconstrained Attribute Class)**

**Example Comparison Function:**

```
bool less(const LabeledPoint& a,
          const LabeledPoint& b) {
  return abs(a.x()) + abs(a.y())
       < abs(b.x()) + abs(b.y());
}
```

**(a.k.a. "Manhattan Distance")**

# What is a *Priority Queue?*

## Example Queue Element:

```
class LabledPoint {
  std::string d_label;
  int          d_x;
  int          d_y;
 public:

  // … (Regular Type)

  const std::string& label() const { return d_label };
  int                x()     const { return d_x; };
  int                y()     const { return d_y; };
};

bool operator==(const LabeledPoint& lhs,
                const LabeledPoint& rhs) {
  return lhs.label() == rhs.label()
      && lhs.x()     == rhs.x()
      && lhs.y()     == rhs.y();
}
```

**(Unconstrained Attribute Class)**

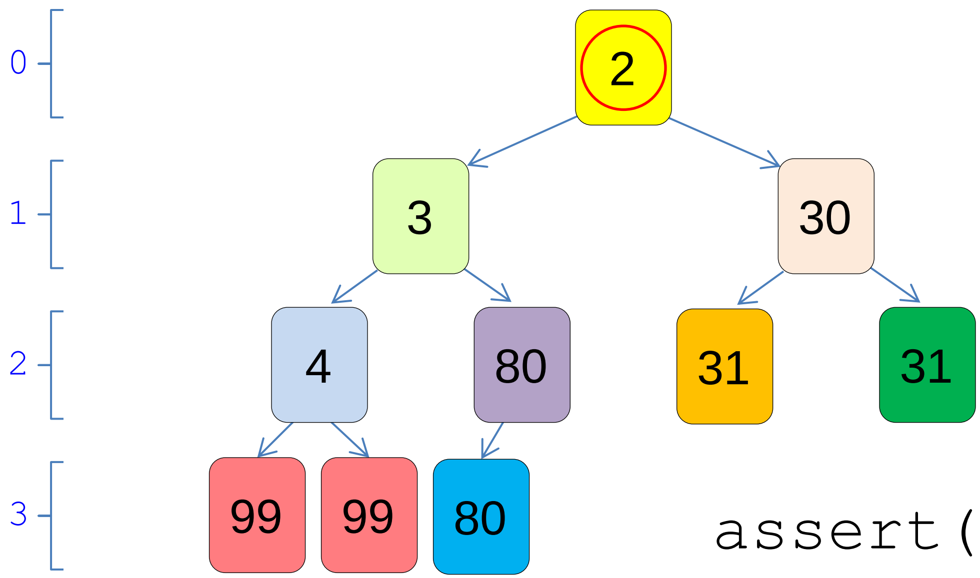
## Example Comparison Function:

```
bool less(const LabeledPoint& a,
          const LabeledPoint& b) {
  return abs(a.x()) + abs(a.y())
       < abs(b.x()) + abs(b.y());
}
```

**(a.k.a. "Manhattan Distance")**

# Priority Queues

## What is a *Priority Queue?*

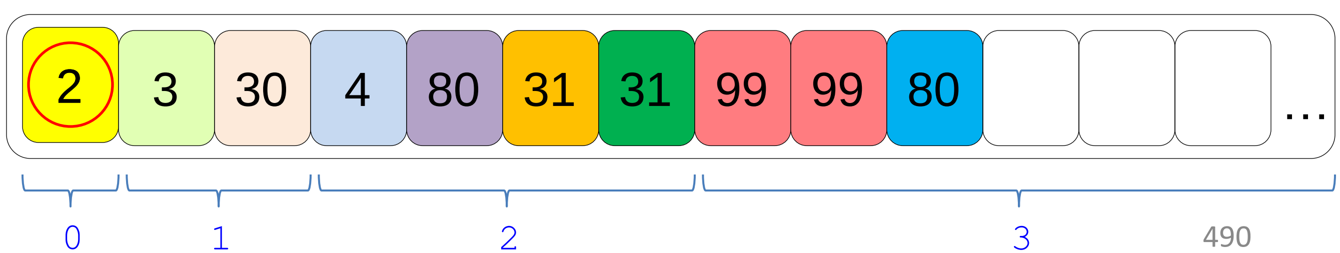



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

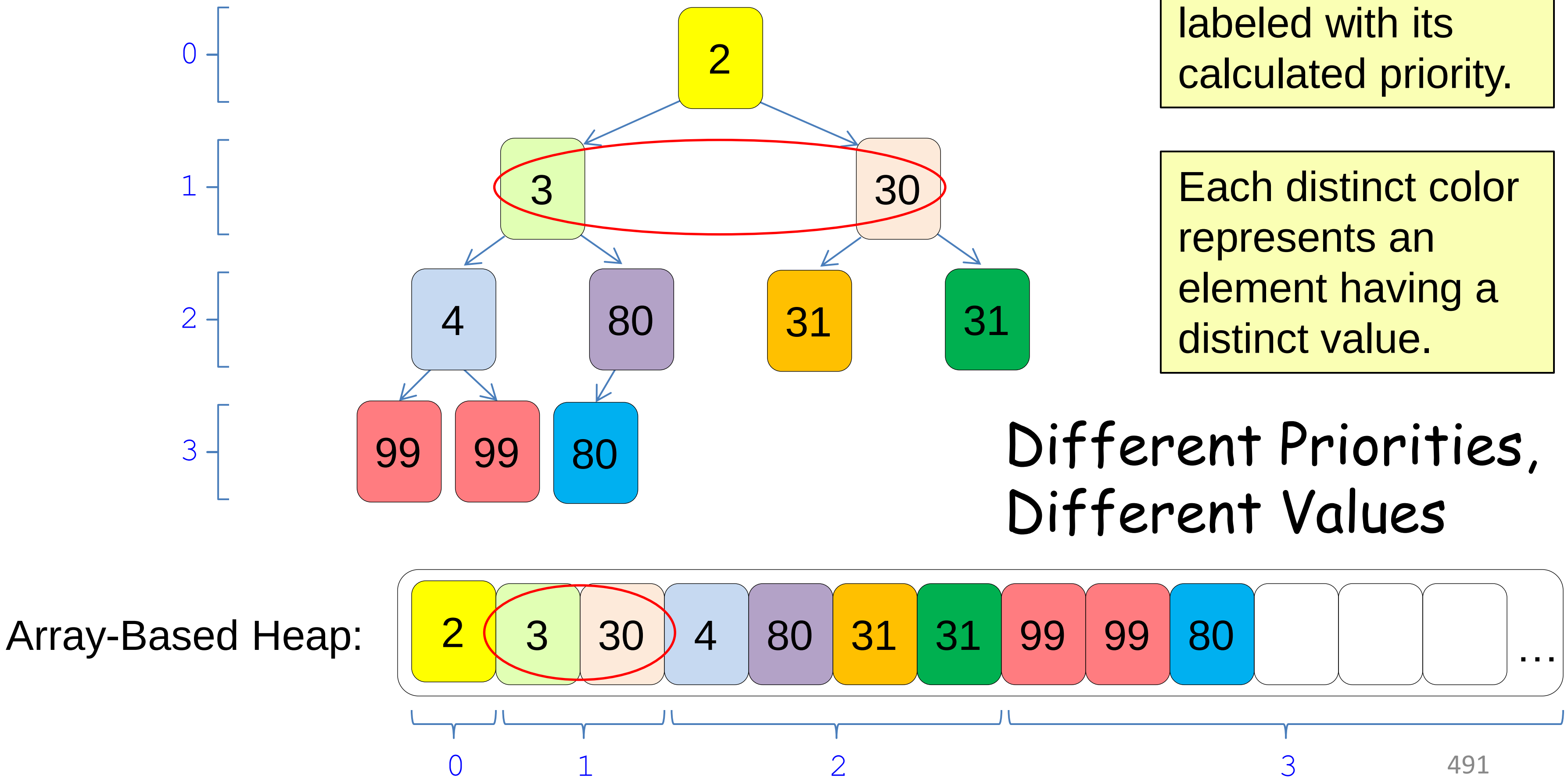
```
assert(q.top() == 2);
```

Array-Based Heap:



# Priority Queues

## What is a *Priority Queue?*



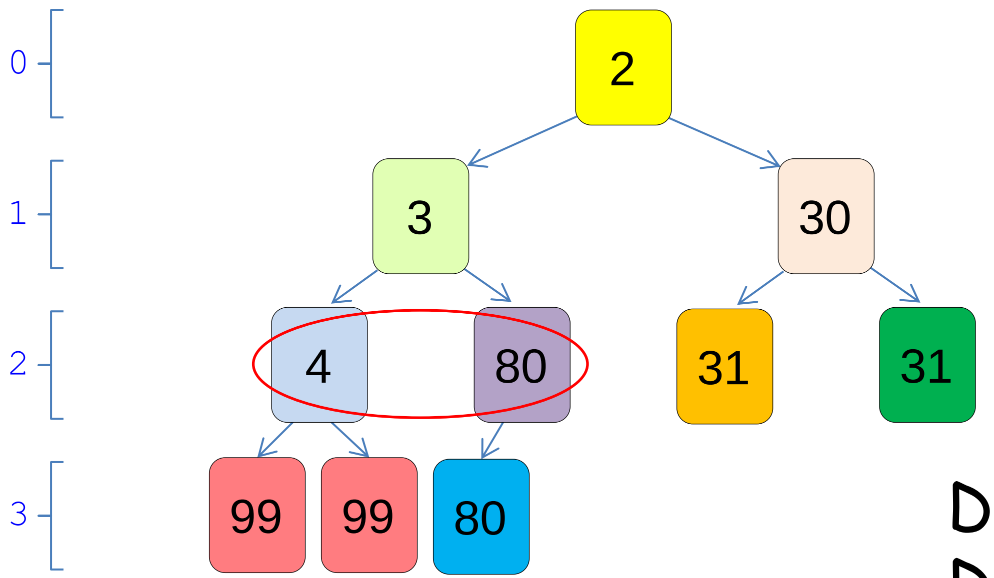

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

Different Priorities, Different Values

# Priority Queues

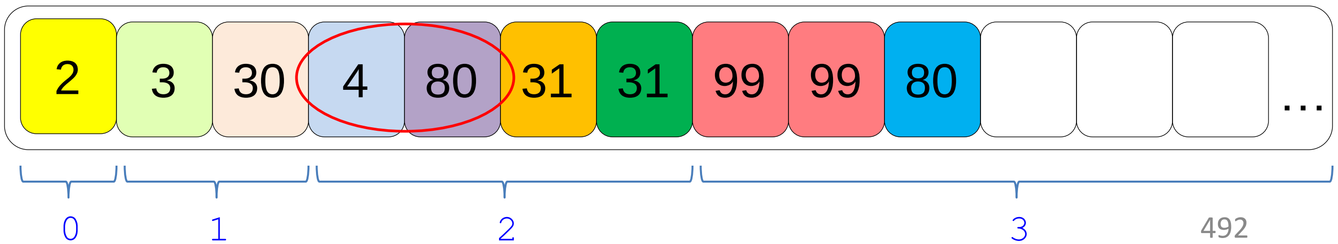
## What is a *Priority Queue?*




Each element is labeled with its calculated priority.

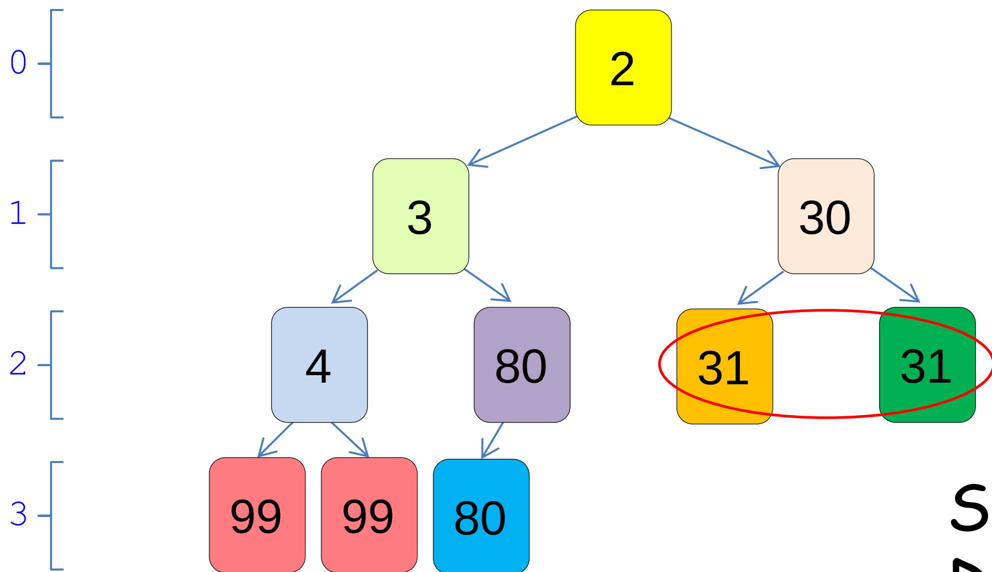
Each distinct color represents an element having a distinct value.

Different Priorities, Different Values



# Priority Queues

## What is a *Priority Queue?*

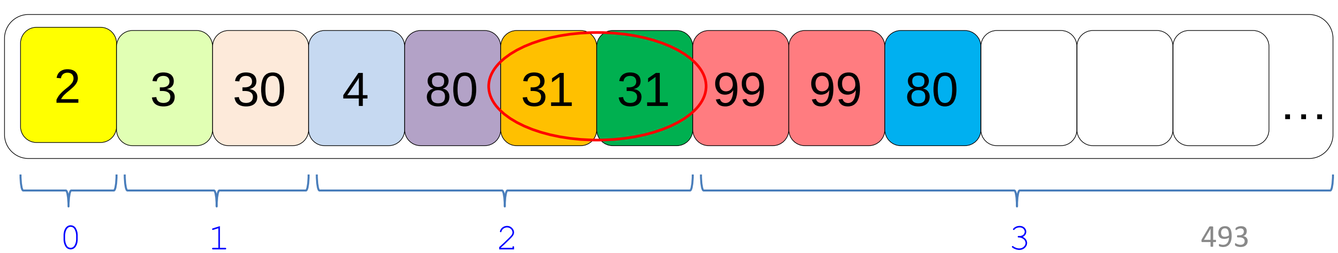



Each element is labeled with its calculated priority.

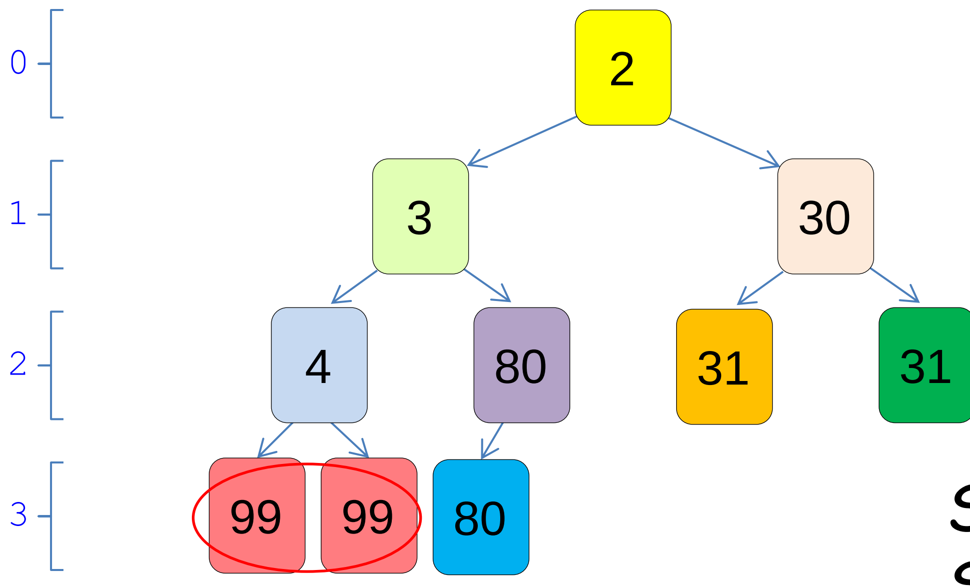
Each distinct color represents an element having a distinct value.

Same Priority, Different Values

Array-Based Heap:



# What is a *Priority Queue?*

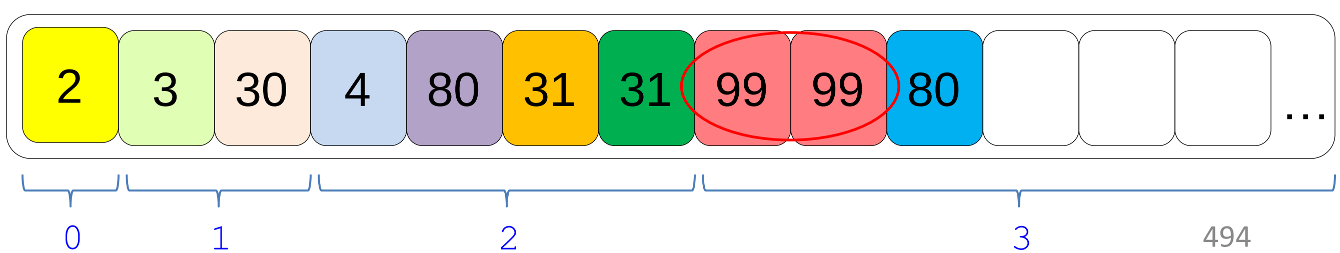



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

Same Priority, Same Value

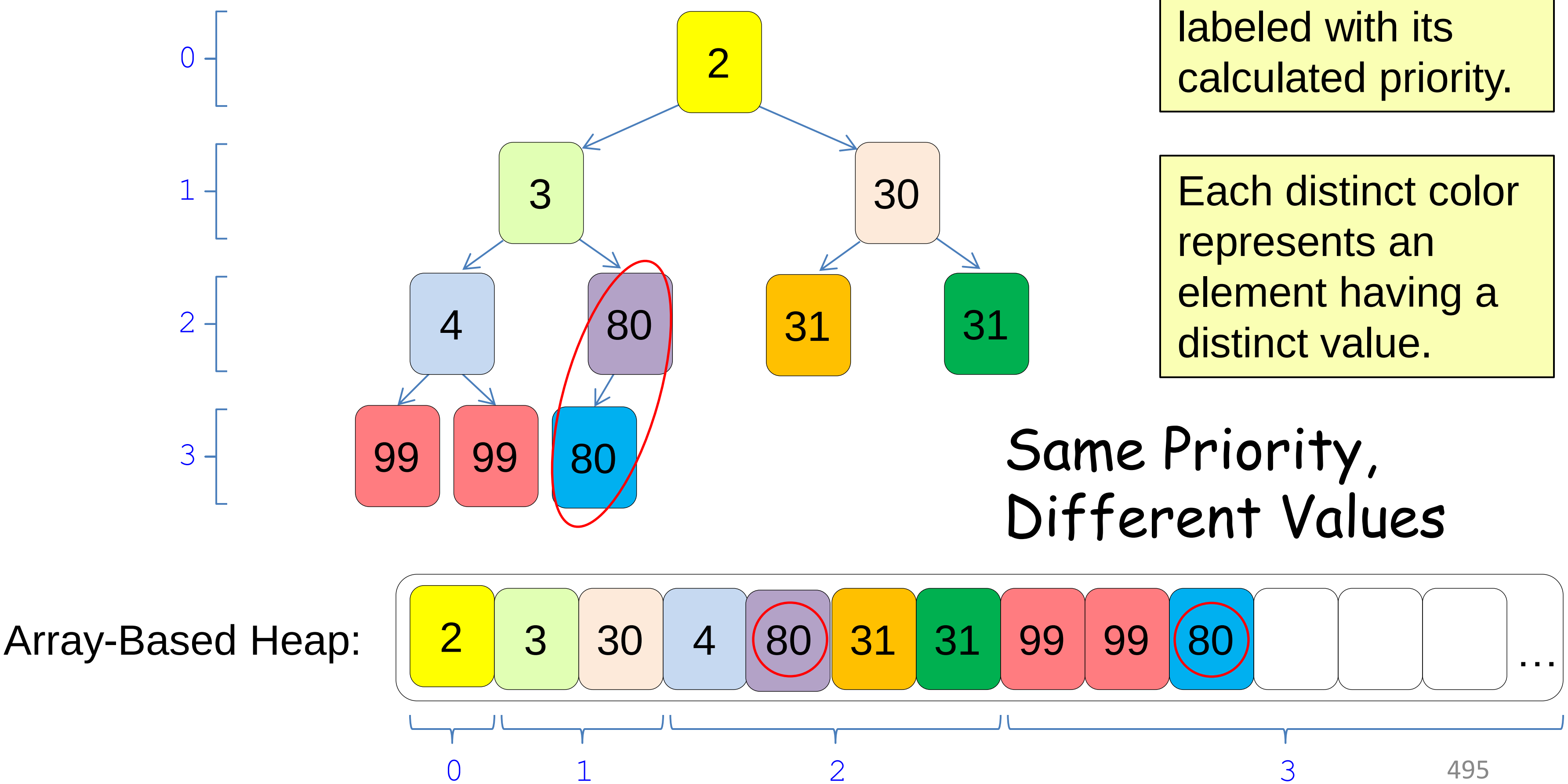
Array-Based Heap:



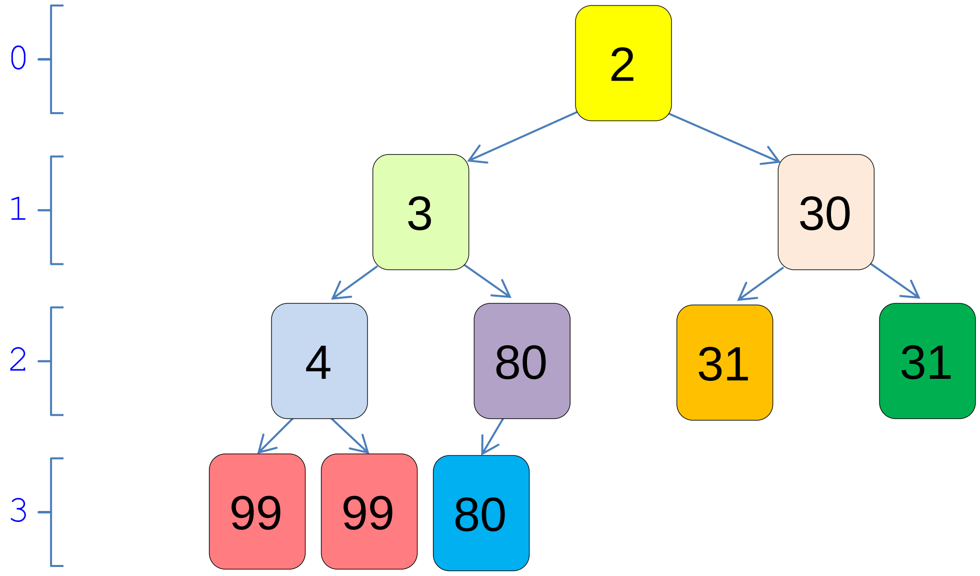
# What is a *Priority Queue?*

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.



# What is a *Priority Queue?*

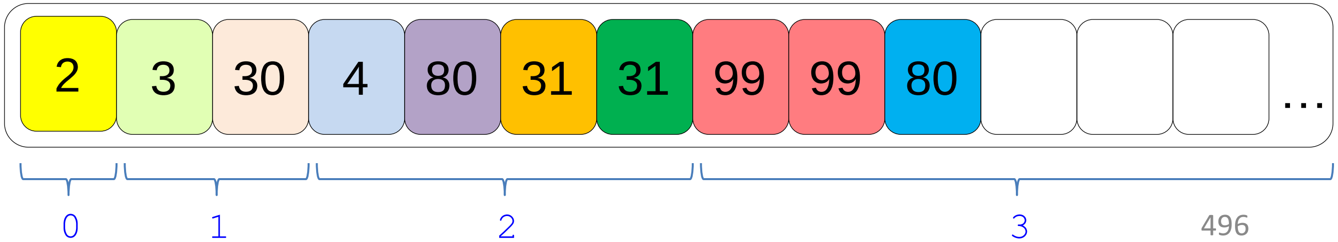



Each element is labeled with its calculated priority.

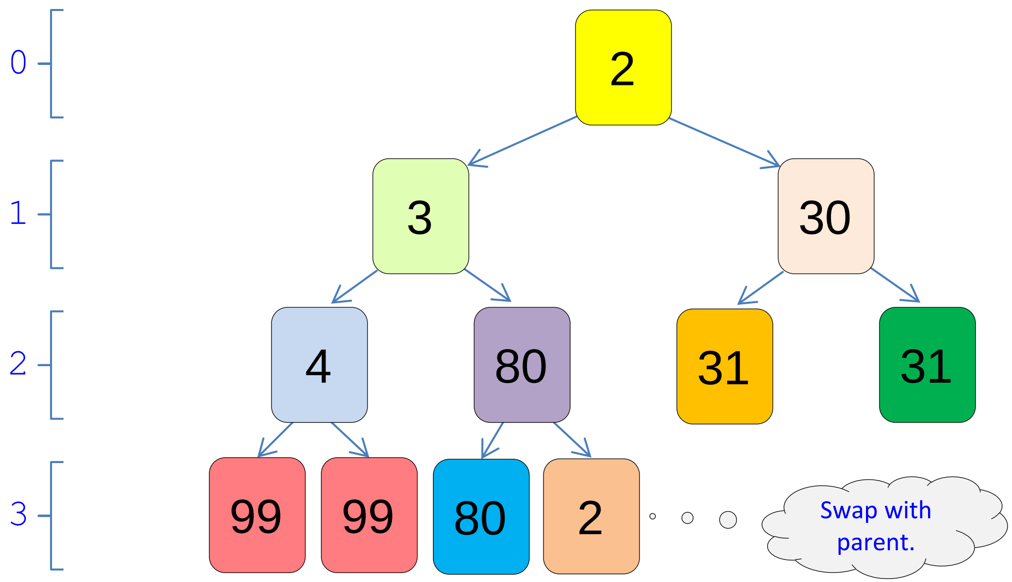
Each distinct color represents an element having a distinct value.

`q.push( 2 );`

Array-Based Heap:



# What is a *Priority Queue?*

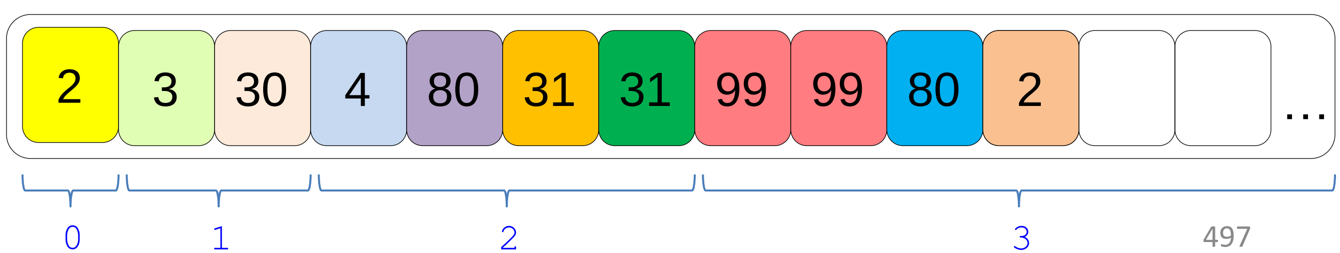



Each element is labeled with its calculated priority.

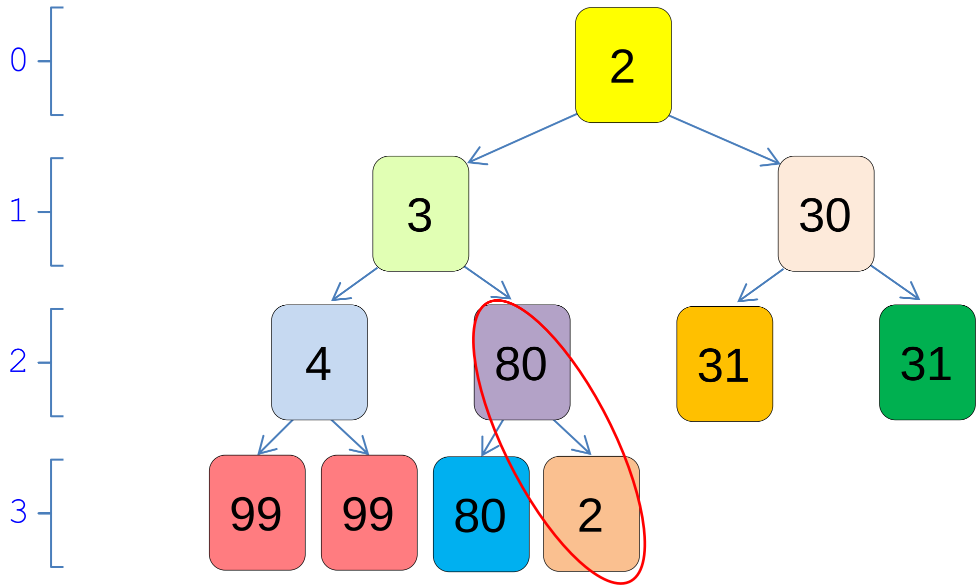
Each distinct color represents an element having a distinct value.

```
q.push( 2 );
```

Array-Based Heap:



# What is a *Priority Queue?*

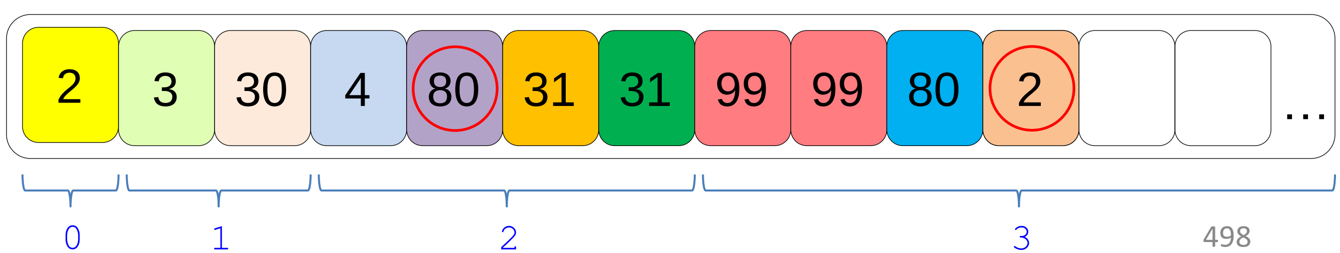



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

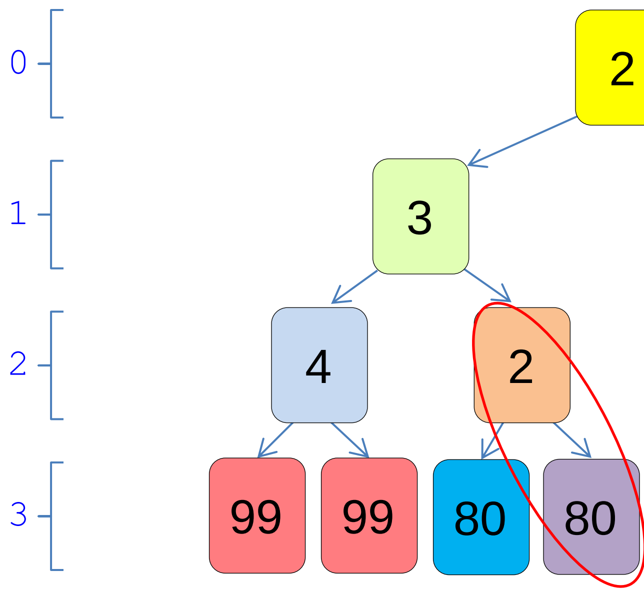
```
q.push( 2 );
```

Array-Based Heap:



# Priority Queues

## What is a *Priority Queue?*

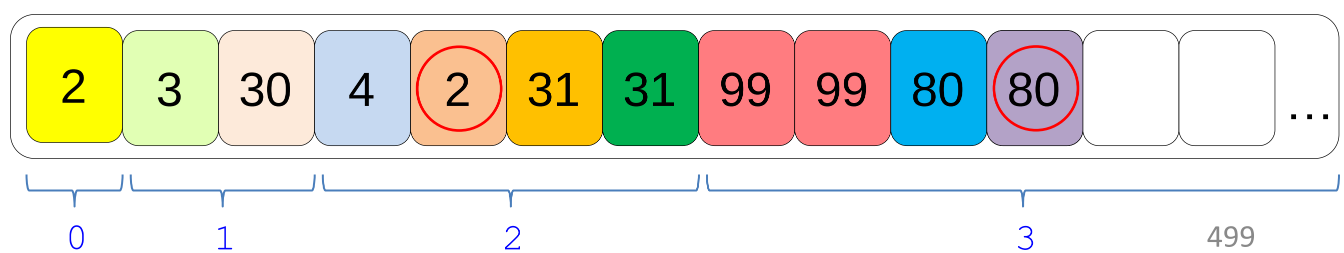



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

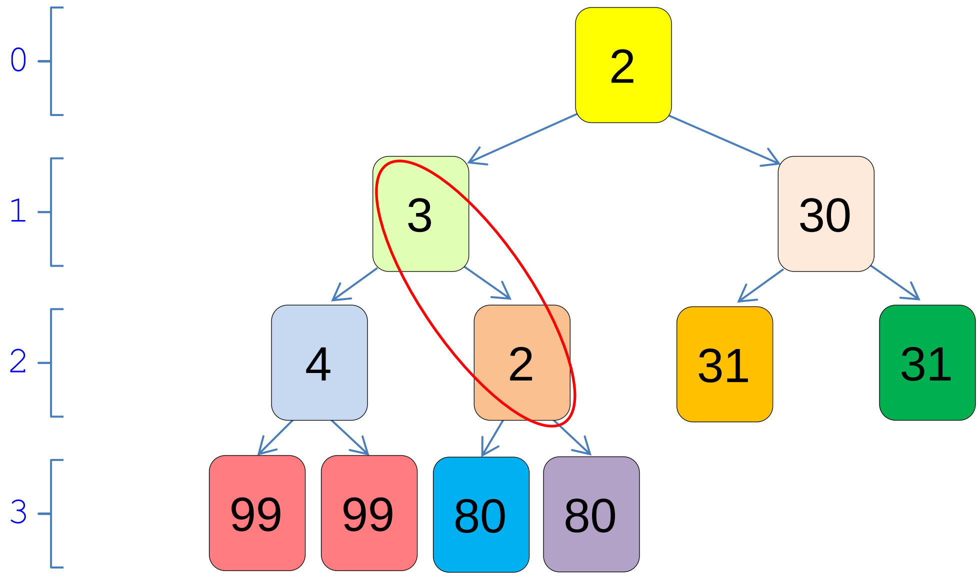
q.push( 2 );

Array-Based Heap:



# Priority Queues

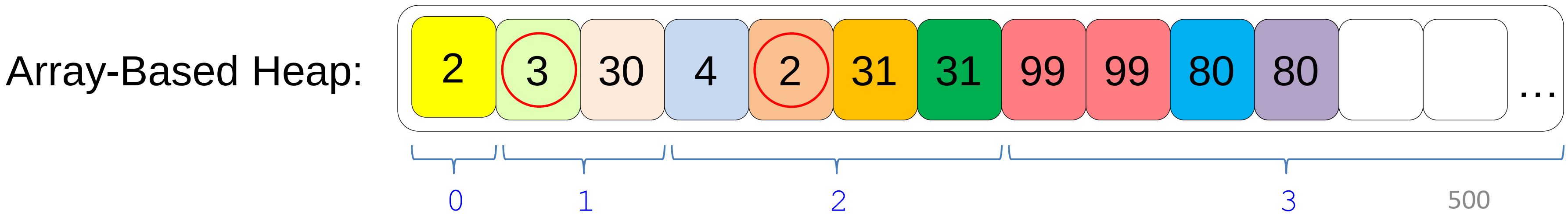
## What is a *Priority Queue?*




Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

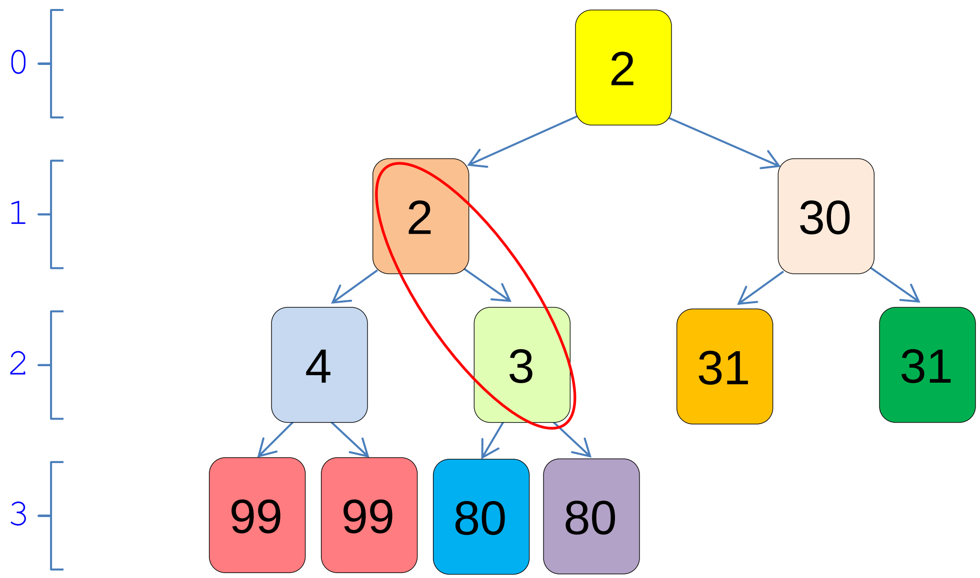
```
q.push( 2 );
```



## 3. Two Important, Instructional Case Studies

# Priority Queues

## What is a *Priority Queue?*

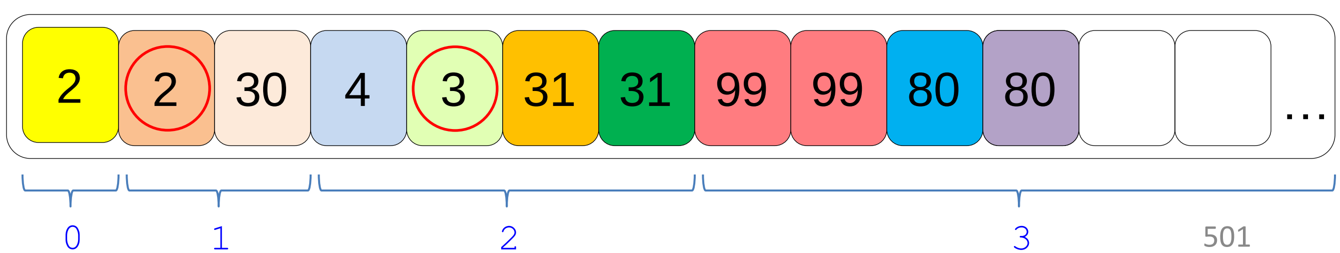



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

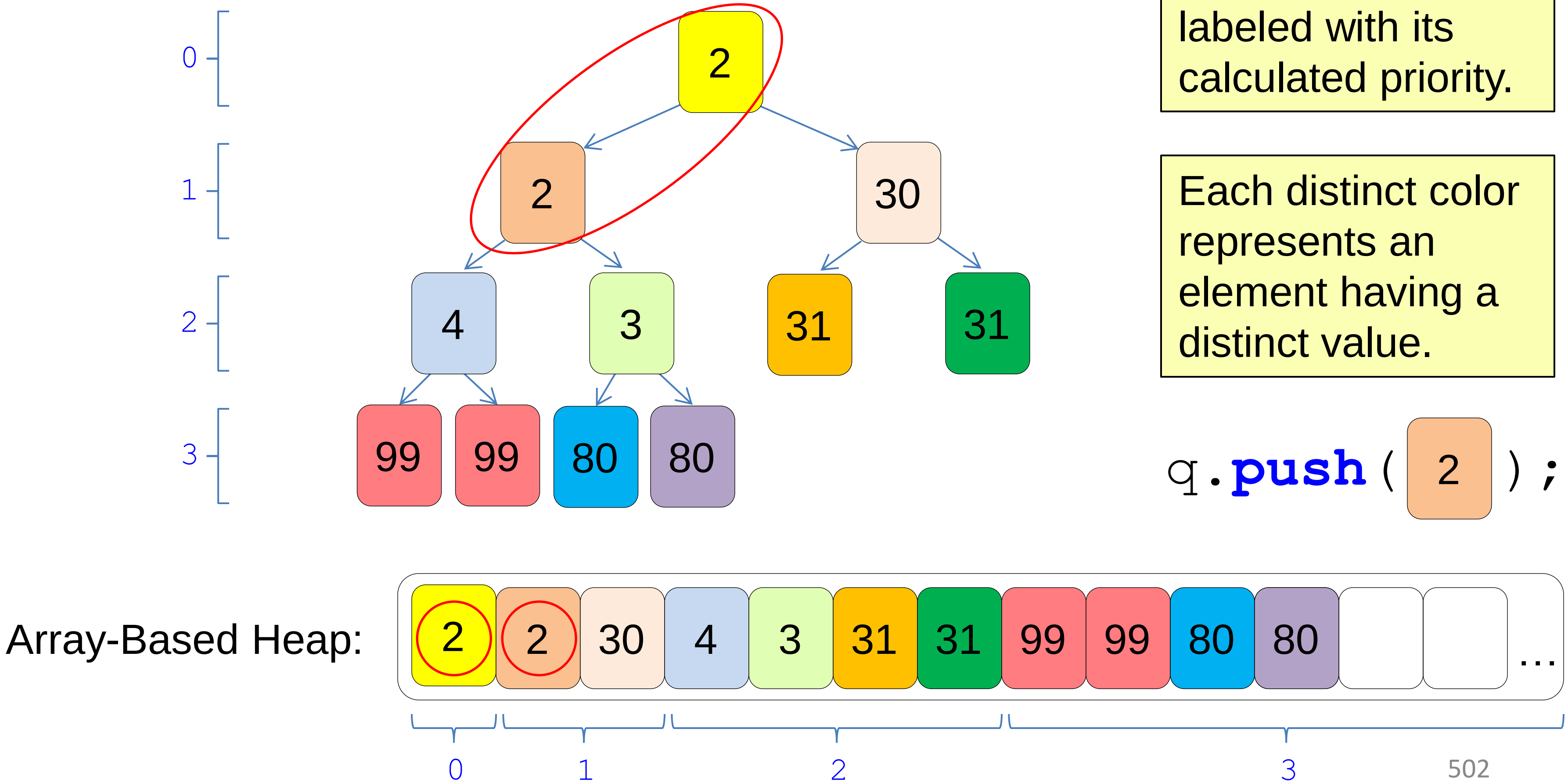
```
q.push( 2 );
```

Array-Based Heap:



# Priority Queues

## What is a *Priority Queue?*

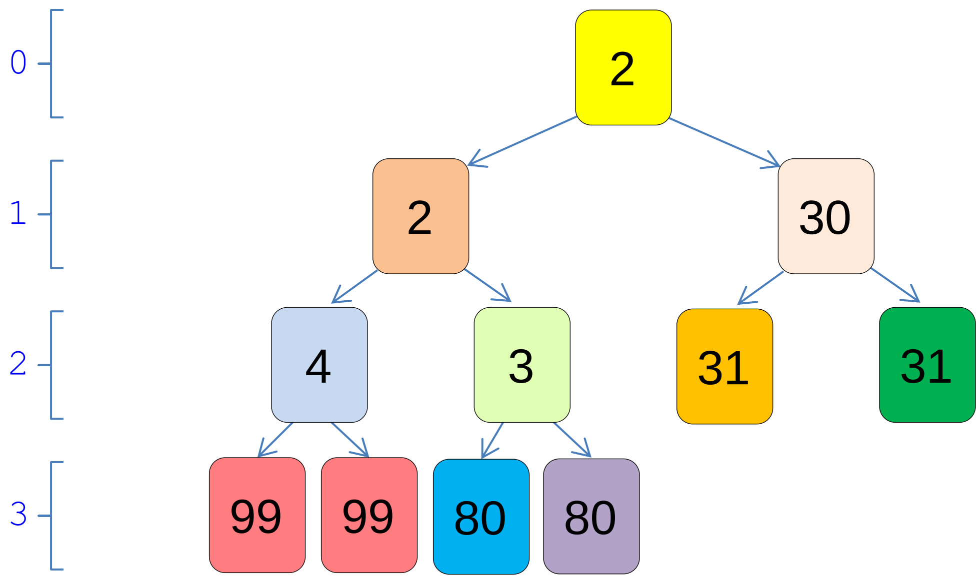



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

`q.push( 2 );`

# What is a *Priority Queue?*

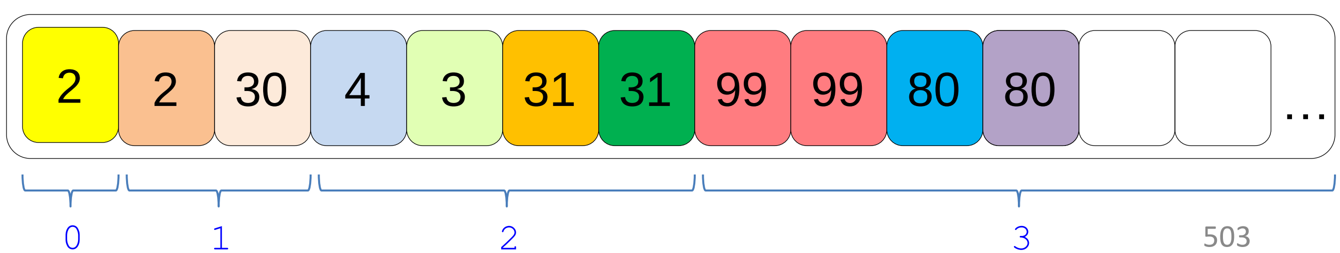



Each element is labeled with its calculated priority.

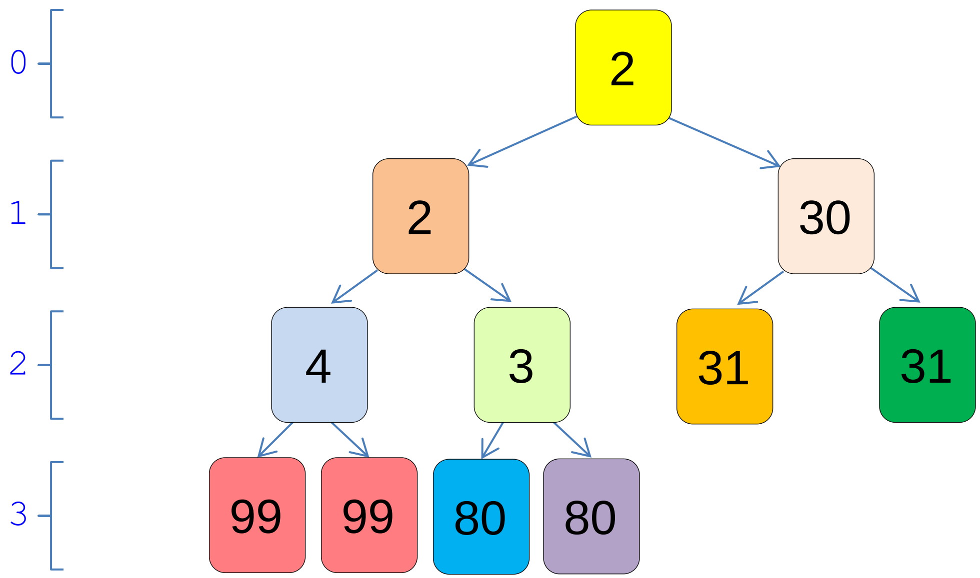
Each distinct color represents an element having a distinct value.

`q.push( 2 );`

Array-Based Heap:



# What is a *Priority Queue?*

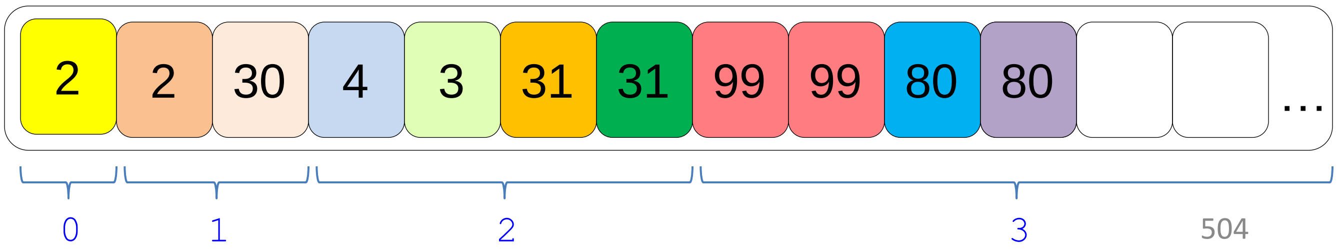



Each element is labeled with its calculated priority.

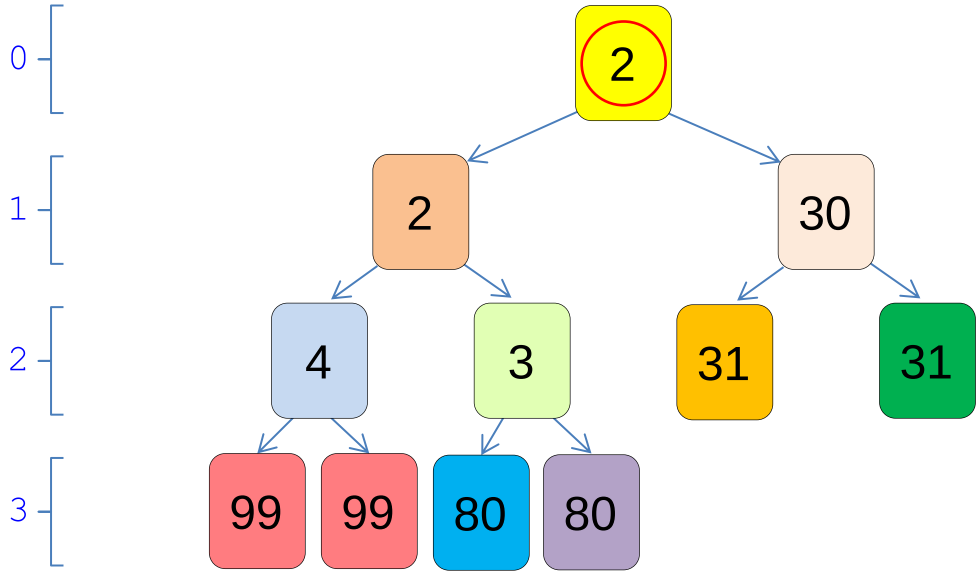
Each distinct color represents an element having a distinct value.

```
q.pop();
```

Array-Based Heap:



# What is a *Priority Queue?*

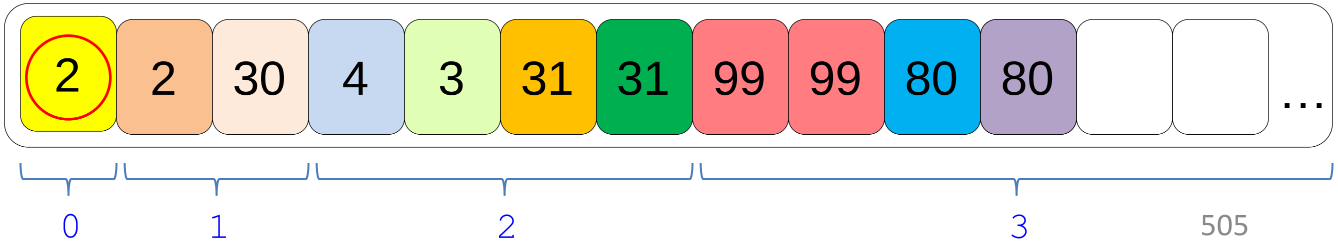



Each element is labeled with its calculated priority.

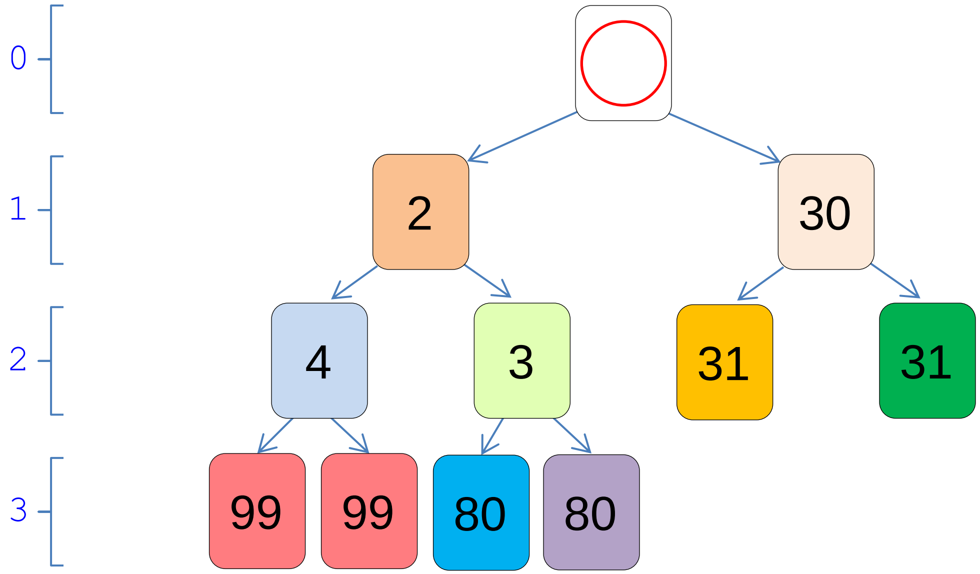
Each distinct color represents an element having a distinct value.

```
q.pop();
```

Array-Based Heap:



# What is a *Priority Queue?*

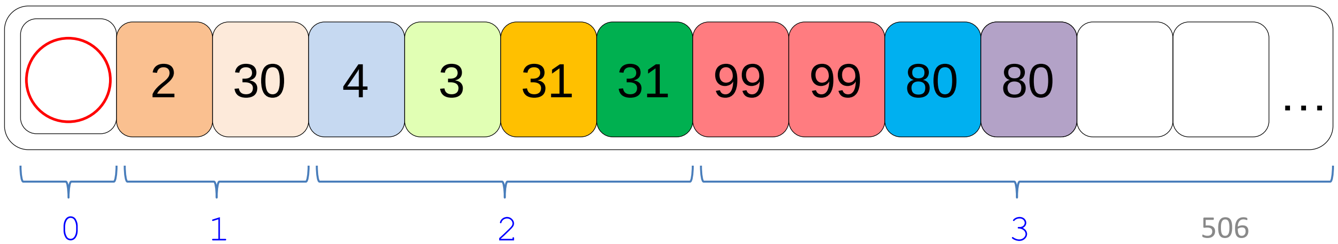



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

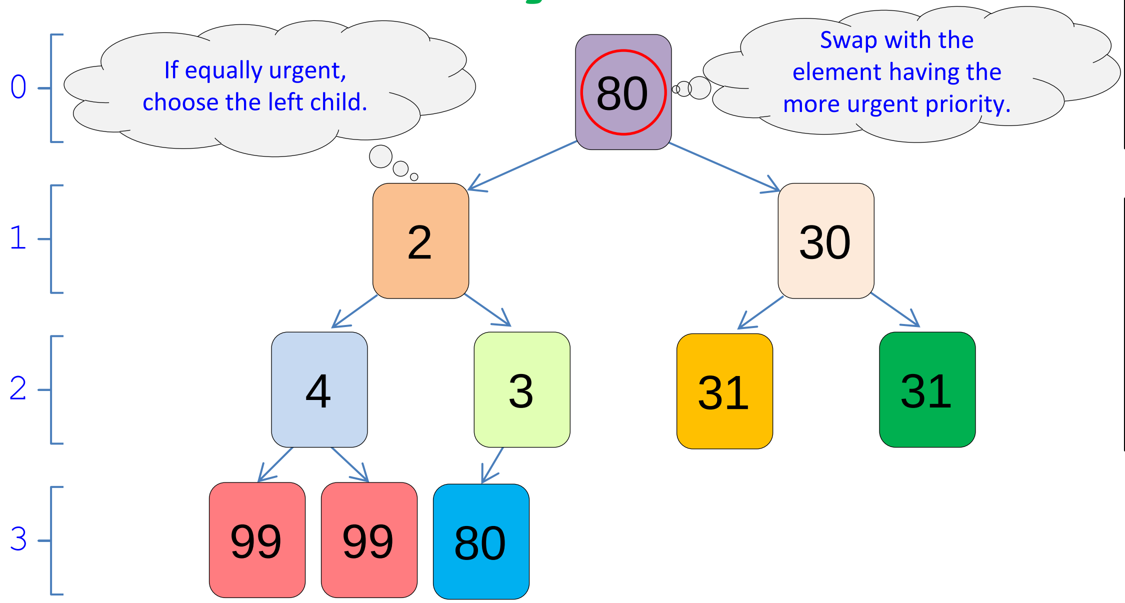
```
q.pop();
```

Array-Based Heap:



# Priority Queues

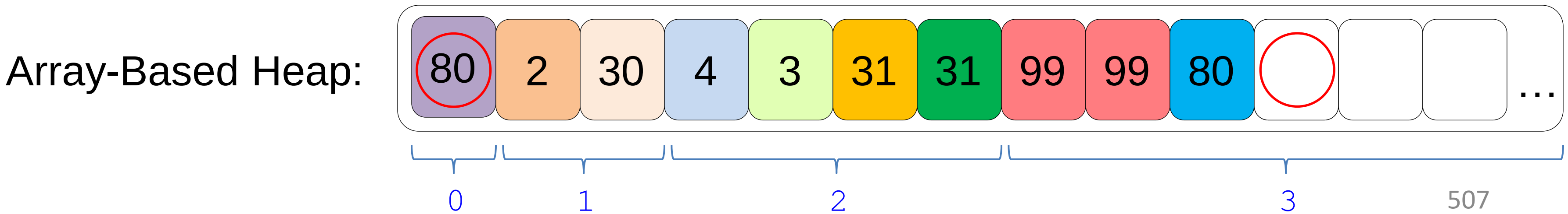
## What is a *Priority Queue?*




Each element is labeled with its calculated priority.

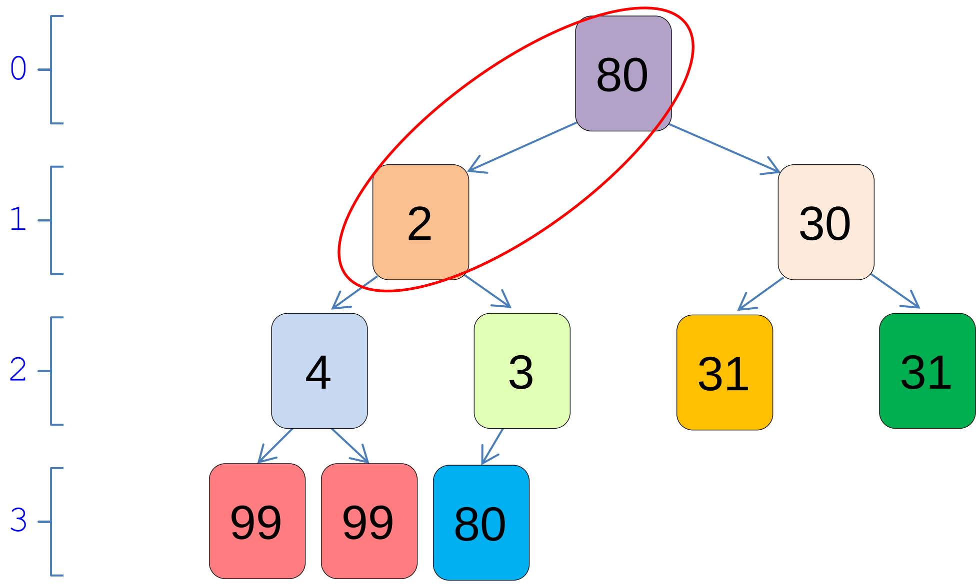
Each distinct color represents an element having a distinct value.

```
q.pop();
```



# Priority Queues

## What is a *Priority Queue?*

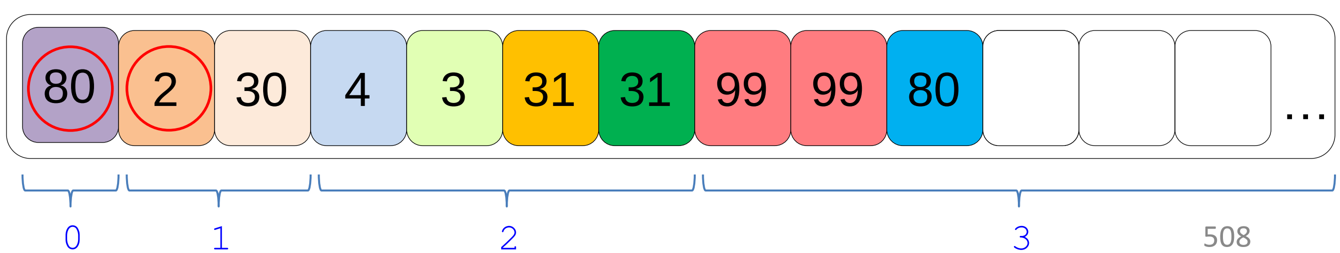



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

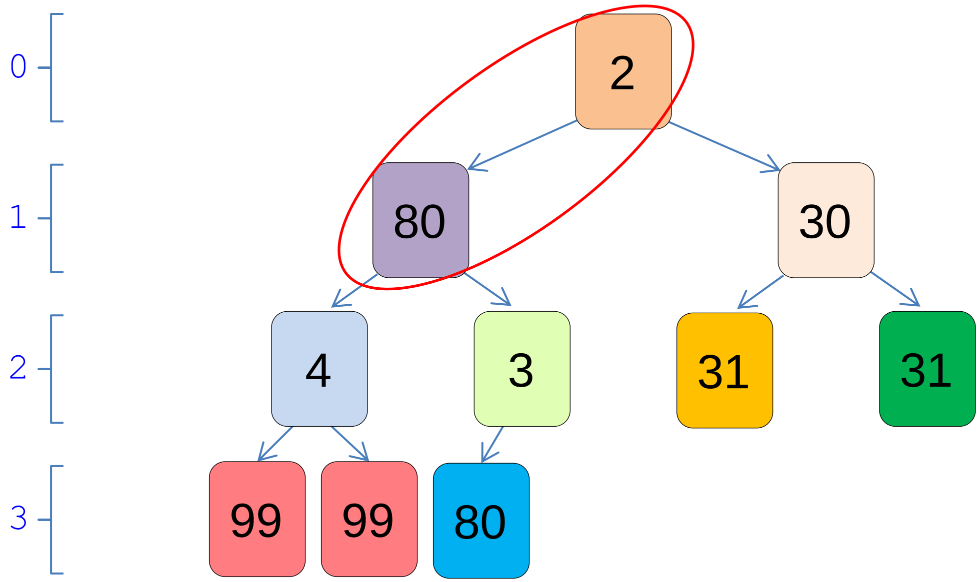
```
q.pop();
```

Array-Based Heap:



# Priority Queues

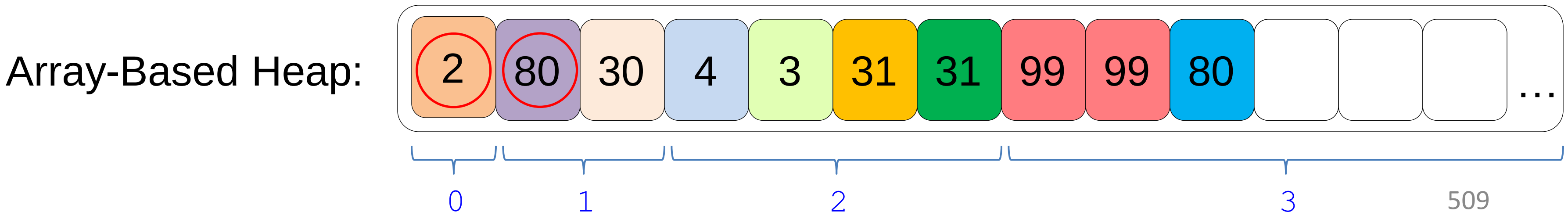
## What is a *Priority Queue?*




Each element is labeled with its calculated priority.

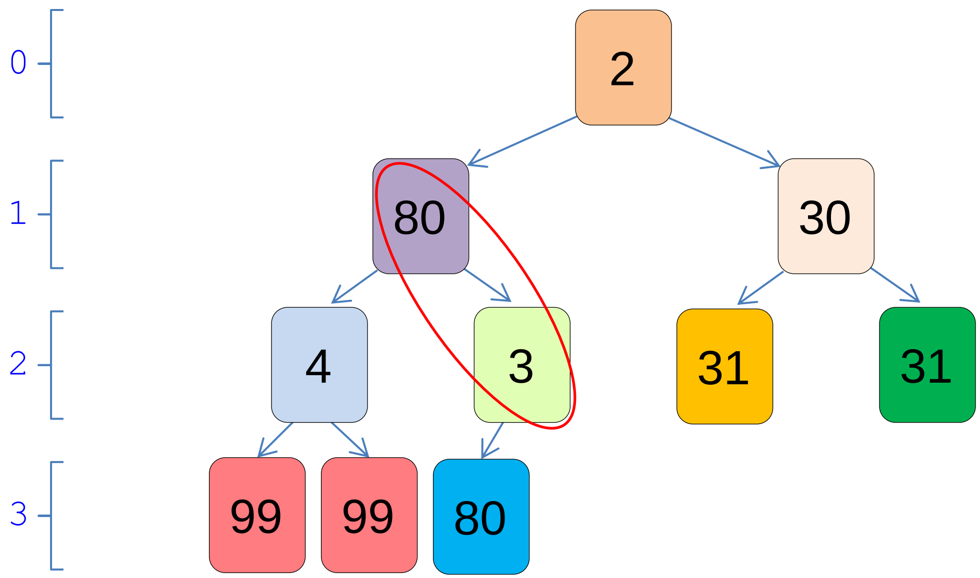
Each distinct color represents an element having a distinct value.

`q.pop();`



# Priority Queues

## What is a *Priority Queue?*

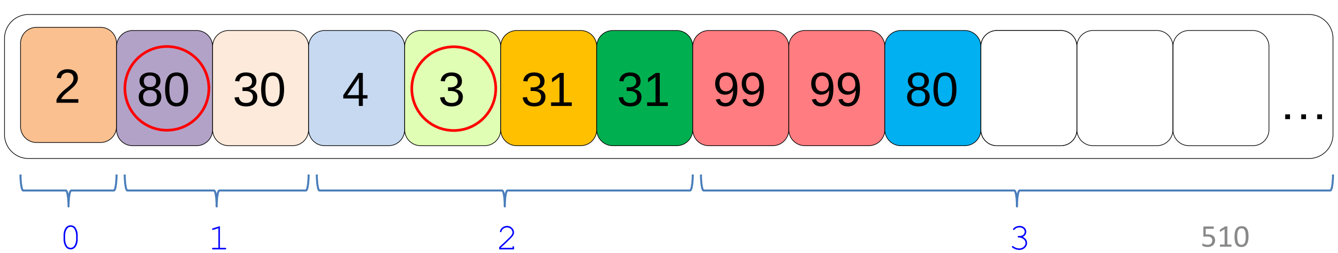



Each element is labeled with its calculated priority.

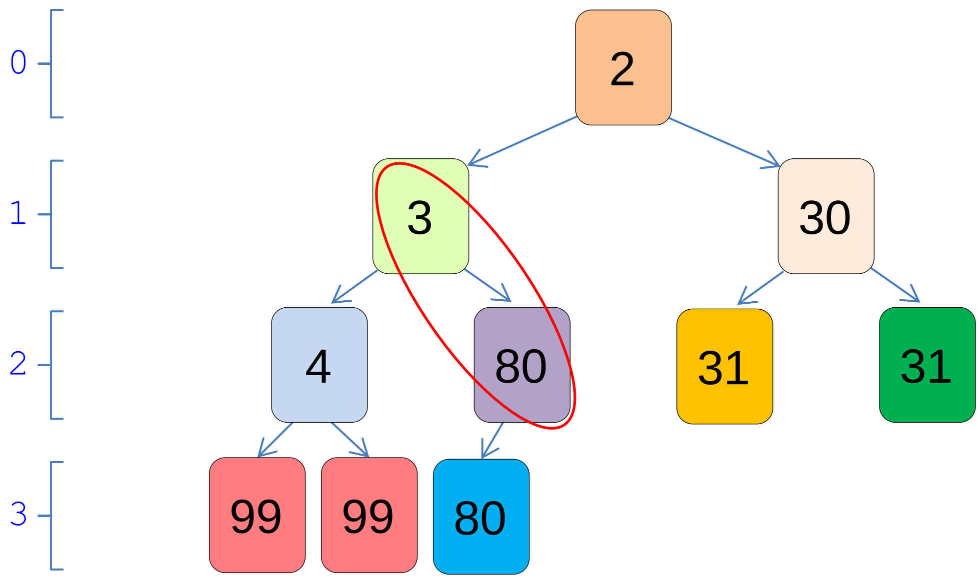
Each distinct color represents an element having a distinct value.

```
q.pop();
```

Array-Based Heap:



# What is a *Priority Queue?*

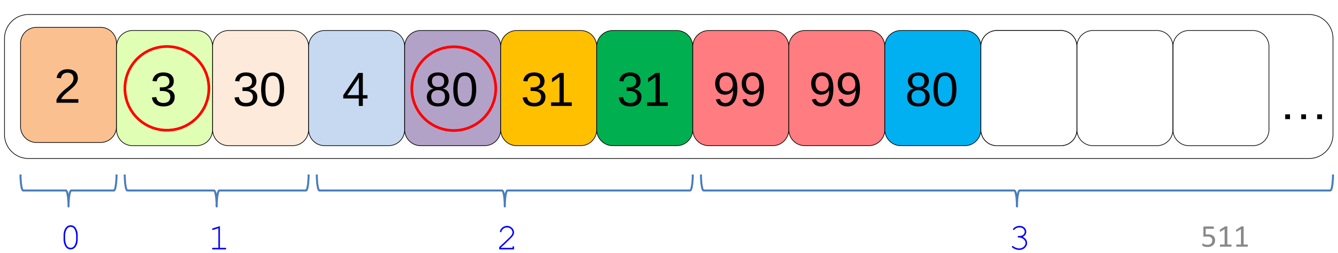



Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

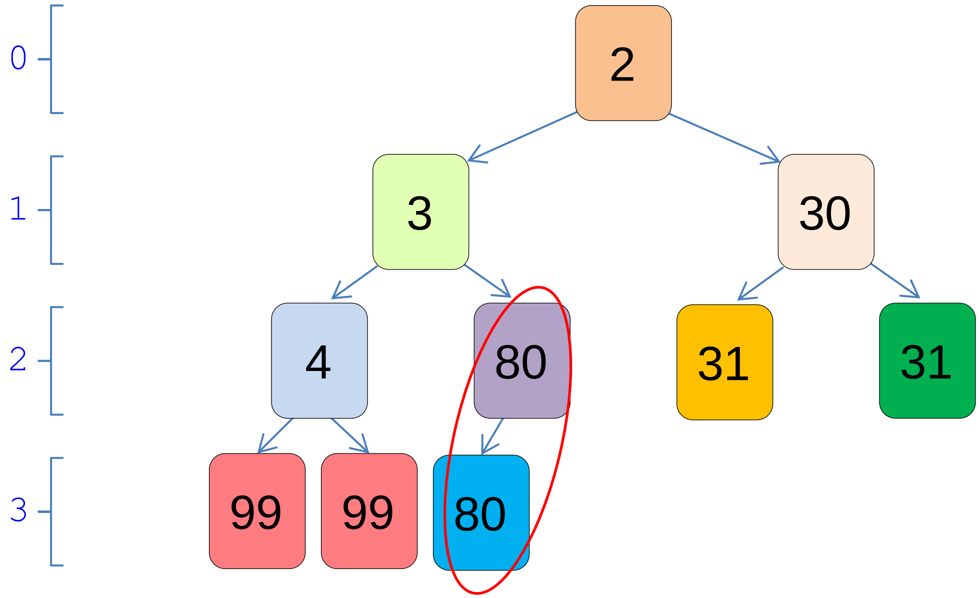
```
q.pop();
```

Array-Based Heap:



# Priority Queues

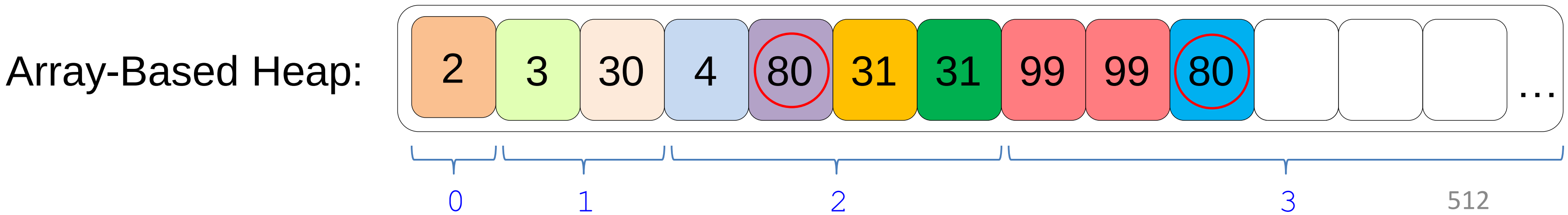
## What is a *Priority Queue?*




Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

```
q.pop();
```

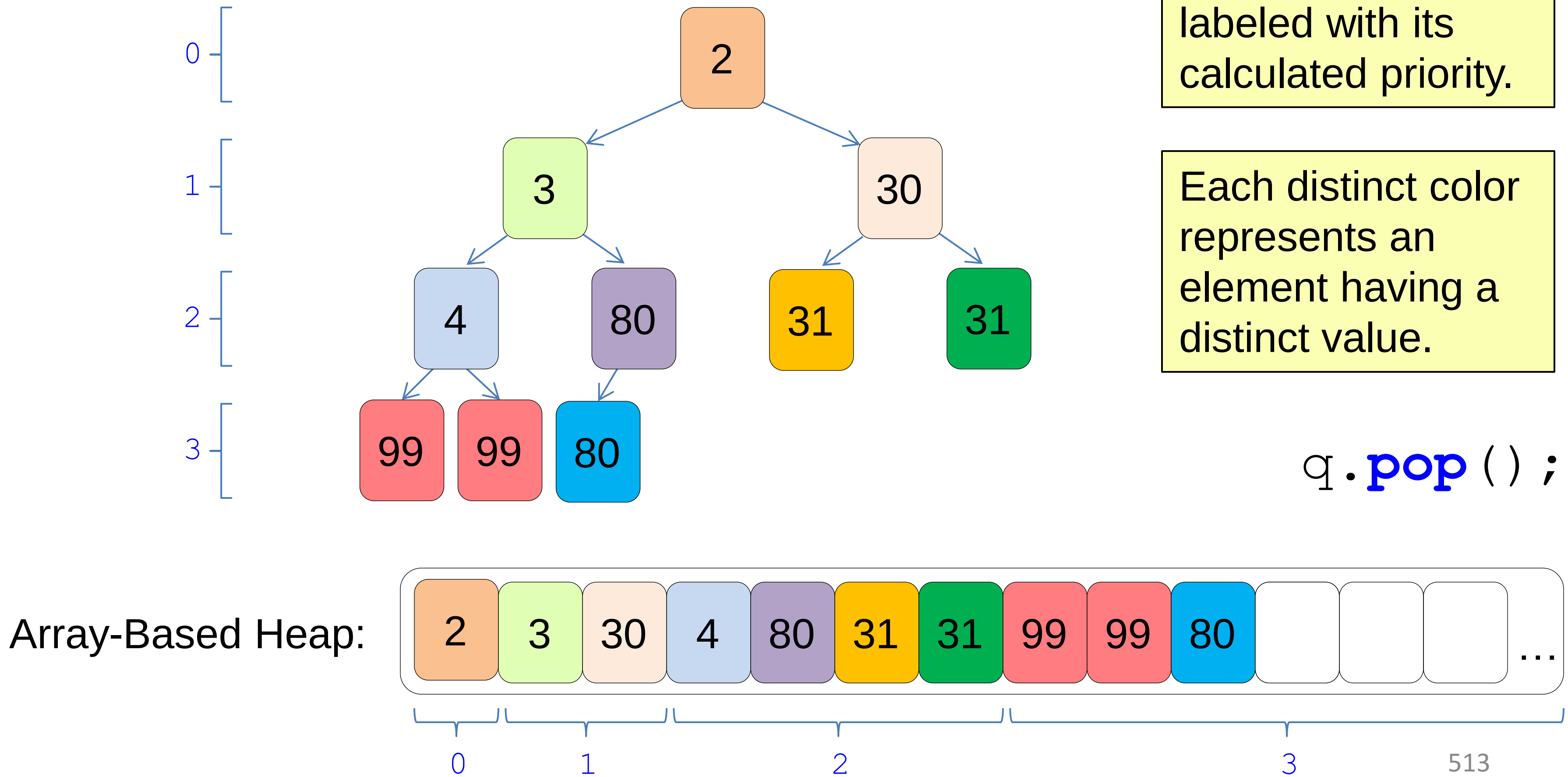


# Priority Queues

## What is a *Priority Queue?*

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.




`q.pop();`

# Priority Queues

## Important Design Questions:

- What is a *Priority Queue*?
- **Why create a separate class for it?**
- Does/should it represent a value?
- How should its value be defined?
- Should such a class be *regular*?

# Priority Queues

## **Why create a separate class for it?**

# Priority Queues

## Why create a separate class for it?

A *Priority Queue* is a useful data structure for dispensing value-semantic (as well as other types of) objects according to a user-specified priority order.

# Priority Queues

Important Design Questions:

- What is a *Priority Queue*?
- Why create a separate class for it?
- **Does/should it represent a value?**
- How should its value be defined?
- Should such a class be *regular*?

# Priority Queues

**Does/should it represent a value?**

# Priority Queues

## Does/should it represent a value?

Is a `PriorityQueue` class a *value type*, or a *mechanism?*

# Priority Queues

## **Does/should it represent a value?**

Is a `PriorityQueue` class a *value type*, or a *mechanism?*

I.e., is there an obvious notion of what it means for two `PriorityQueue` objects to have the same value?

# Priority Queues

## Does/should it represent a value?

**I claim, "yes!"**

I.e., is there an obvious notion of what it means for two `PriorityQueue` objects to have the same value?

# Priority Queues

## Does/should it represent a value?

**I claim, “yes!”**

I.e., is th… of what it means f… eue objects t… ?

Assuming, of course, that the queue-element type is also value semantic.

# Priority Queues

## Important Design Questions:

- What is a *Priority Queue*?
- Why create a separate class for it?
- Does/should it represent a value?
- **How should its value be defined?**
- Should such a class be *regular*?

# Priority Queues

**How should its value be defined?**

# Priority Queues

**How should its value be defined?**

# Priority Queues

## How should its value be defined?

# Priority Queues

## How should its value be defined?

Two objects of class `PriorityQueue` have the same value iff there does <u>not</u> exist a *distinguishing sequence* among all of its *salient* operations:

1. `top`
2. `push`
3. `pop`

# Priority Queues

Important Design Questions:

- What is a *Priority Queue*?
- Why create a separate class for it?
- Does/should it represent a value?
- How should its value be defined?
- **Should such a class be *regular*?**

# Priority Queues

## **Should such a class be regular?**

I.e., should our `PriorityQueue` class support all of the value-semantic syntax of a *regular* class?

# Priority Queues

**Should such a class be regular?**

I.e., should our `PriorityQueue` class support all of the value-semantic syntax of a *regular* class?

Question: How *expensive* would `operator==` be to implement?

## Priority Queues

# **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

Moreover, how on earth would we determine whether two arbitrary `PriorityQueue` objects do or do not have a *distinguishing sequence* of *salient* operations??

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

Necessary:

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

# Necessary:

- Same number of elements.

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

## Necessary:

- Same number of elements.
- Same numbers of respective element values.

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

Necessary:

- Same number of elements.
- Same numbers of respective element values.

Sufficient:

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

# Necessary:

- Same number of elements.
- Same numbers of respective element values.

# Sufficient:

- Same underlying linear heap order.

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

# Necessary:

- Same number of elements.
- Same numbers of respective element values.

# Sufficient:

- Same underlying linear heap order.

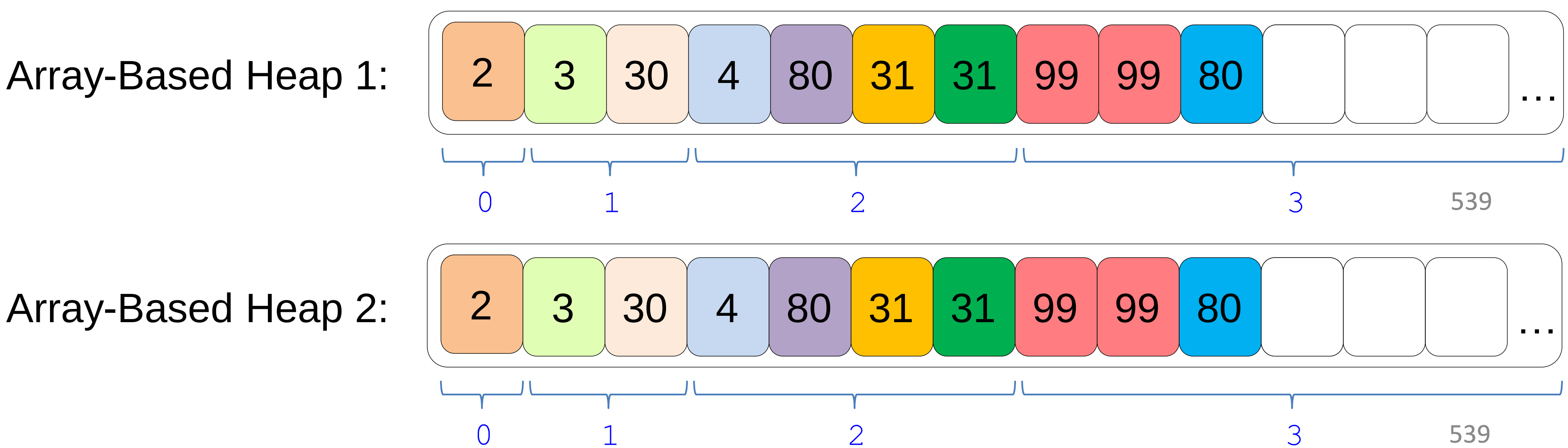
**BUT IS THIS NECESSARY OR NOT??**

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

For example, both of these linear heaps pop in the same order:

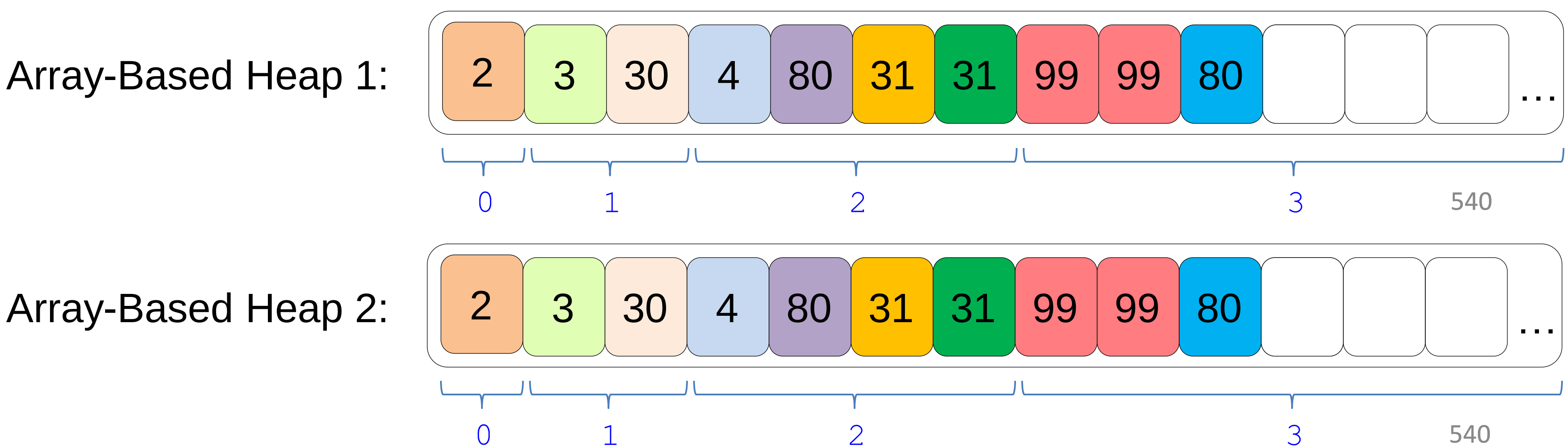


# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

For example, both of these linear heaps pop in the same order (**of course!**):



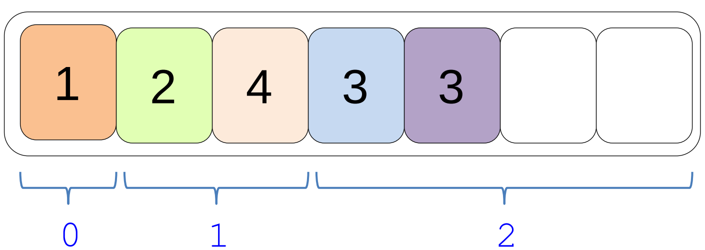
# Priority Queues

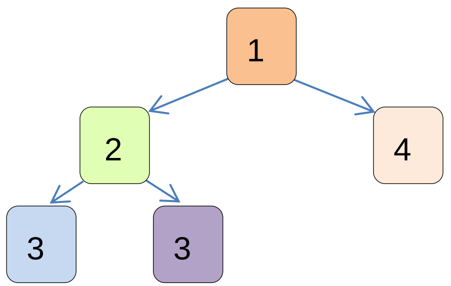
## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

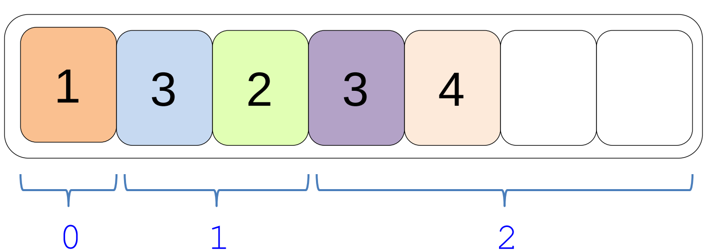
But so do these:

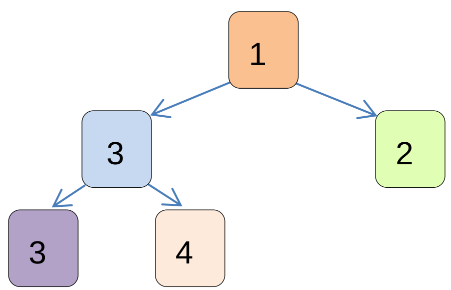
Array-Based Heap 1:




Array-Based Heap 2:



# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

As it turns out, we can distinguish these two values with appropriate pushes, tops, and pops.

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

As it turns out, we can distinguish these two values with appropriate pushes, tops, and pops.

But can we always do that?

## Priority Queues

# **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

As it turns out, we can distinguish these two values with appropriate pushes, tops, and pops.

But can we always do that?

If we aren't sure, should we implement `operator==` for this class anyway?

# Priority Queues

## Should [illegible]

Qu[illegible]

What if we know that more than 99.99% (but less than 100%) of the time we can distinguish the values of two `PriorityQueue` objects that do <u>not</u> have the same linear heap orderings?

[illegible]

But [illegible]n we always do that?

If we aren't sure, should we implement `operator==` for this class anyway?

# Priority Queues

## Should such

Qu

What if we know th

99.99% (b

orderings?

top

But … n we alw

**I say No!**

**Perhaps make it a named function.**

If we aren’t sure, should we implement `operator==` for this class anyway?

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

Suppose it were true that, for any pair of priority queues, where the linear heap order is not the same, there exists a distinguishing sequence of salient operations that distinguishes them:

**What is the complexity of `operator==`?**

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

Suppose it were true that, for any pair of priority qu… linear heap order … here exists a distinguish… of salient operations that distinguishes them:

O[n]

**What is the complexity of `operator==`?**

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

Until quite recently, that linear order *is necessary* was just a conjecture.

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

Until quite recently, that linear order ***is necessary*** was just a conjecture.

I finally have a simple constructive proof.

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

Until quite recently, that linear order *is necessary* was just a conjecture.

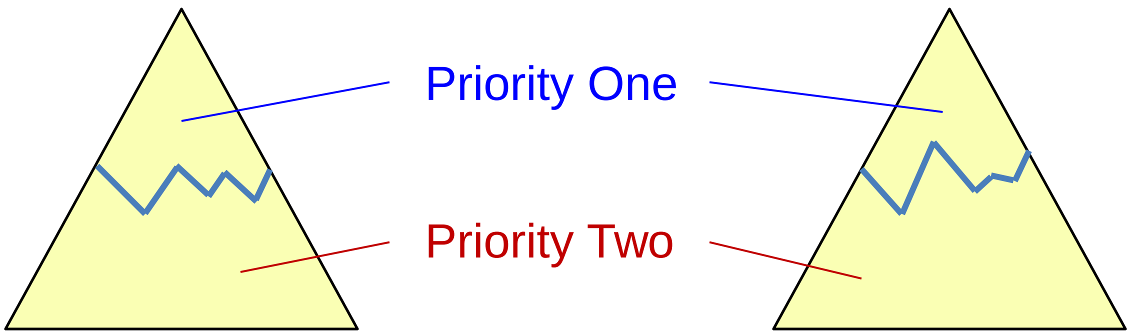
I finally have a simple constructive proof.

Here is a ***very*** quick sketch:

# Priority Queues

## Should such a class be regular?

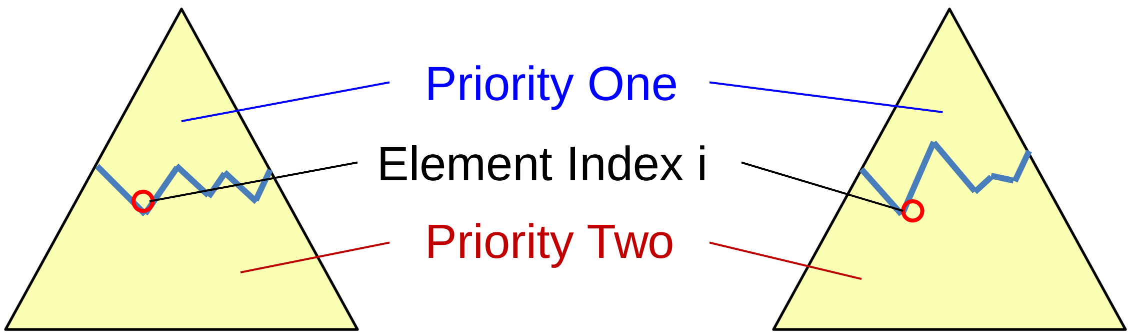
Question: How *expensive* would `operator==` be to implement?



# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

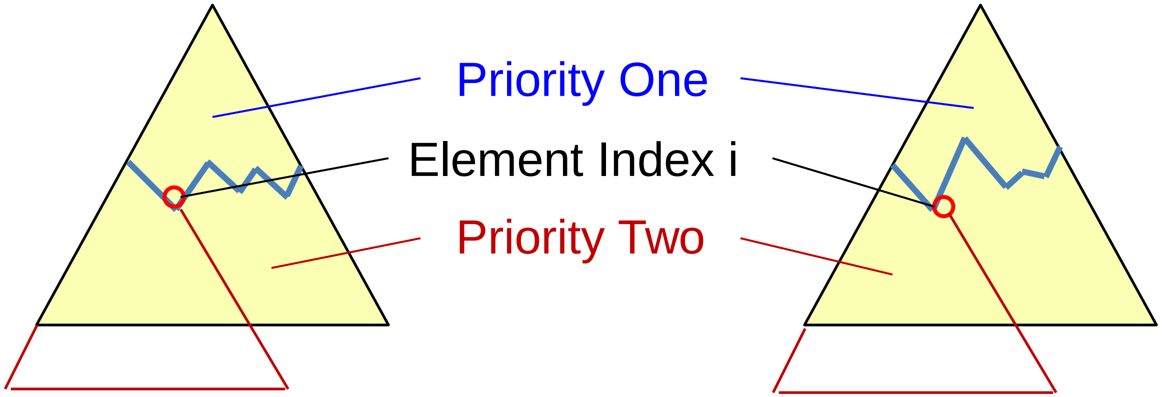



Highest-Index Element Having Distinct Priorities

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

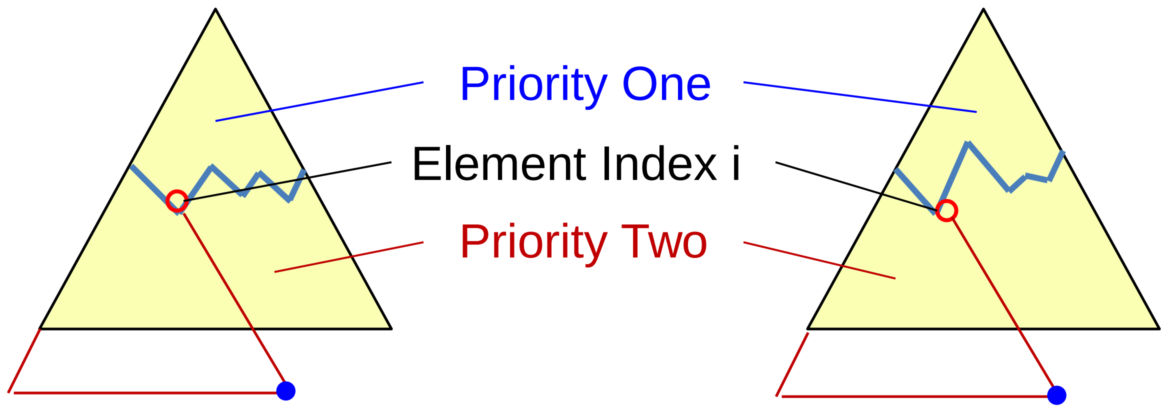



Push Arbitrary Priority-Two Values

# Priority Queues

# **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

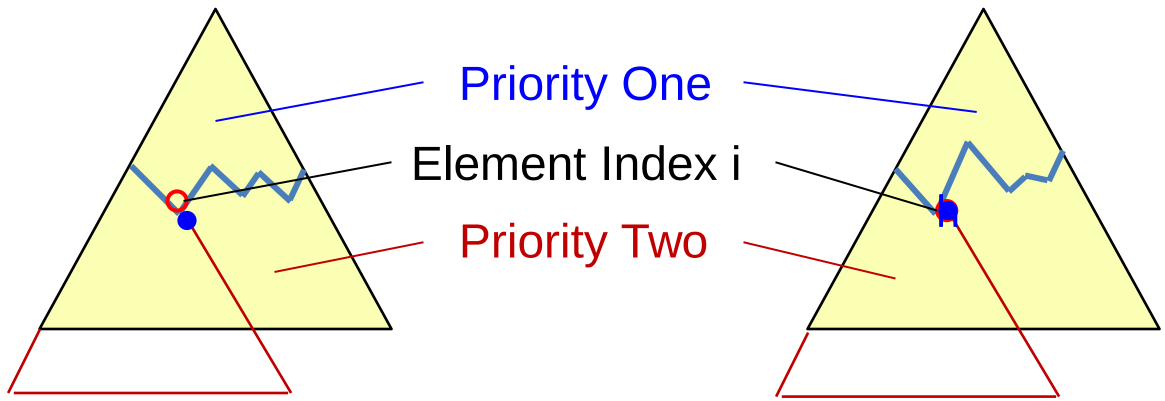



Push a Priority-One Value

# Priority Queues

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

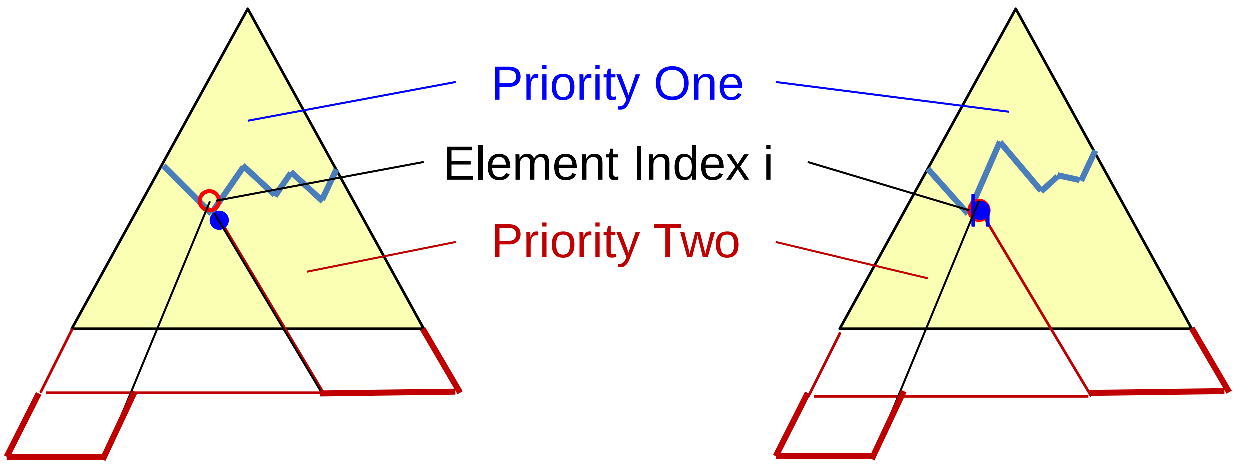



## Push a Priority-One Value

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

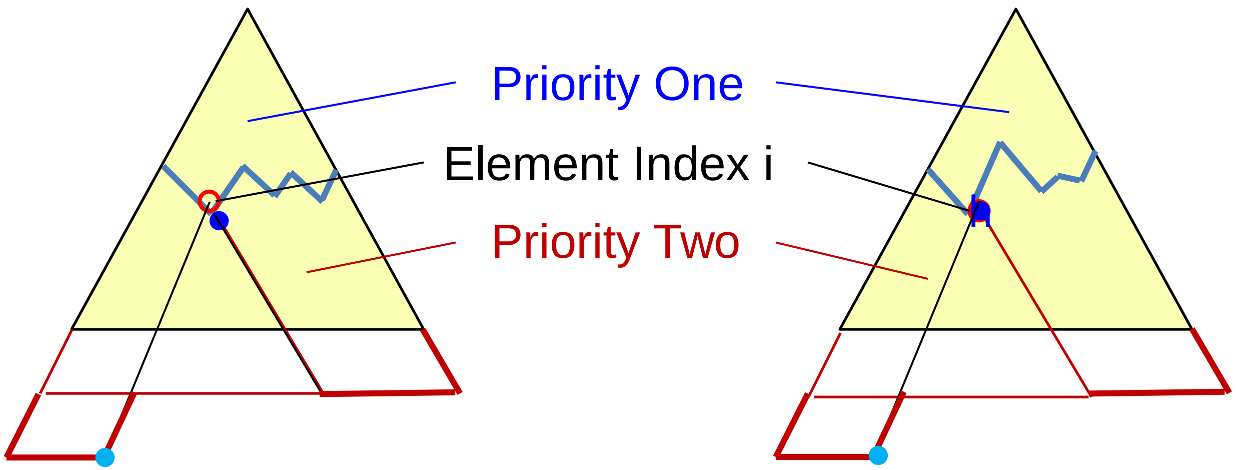



Push Arbitrary Priority-Two values

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

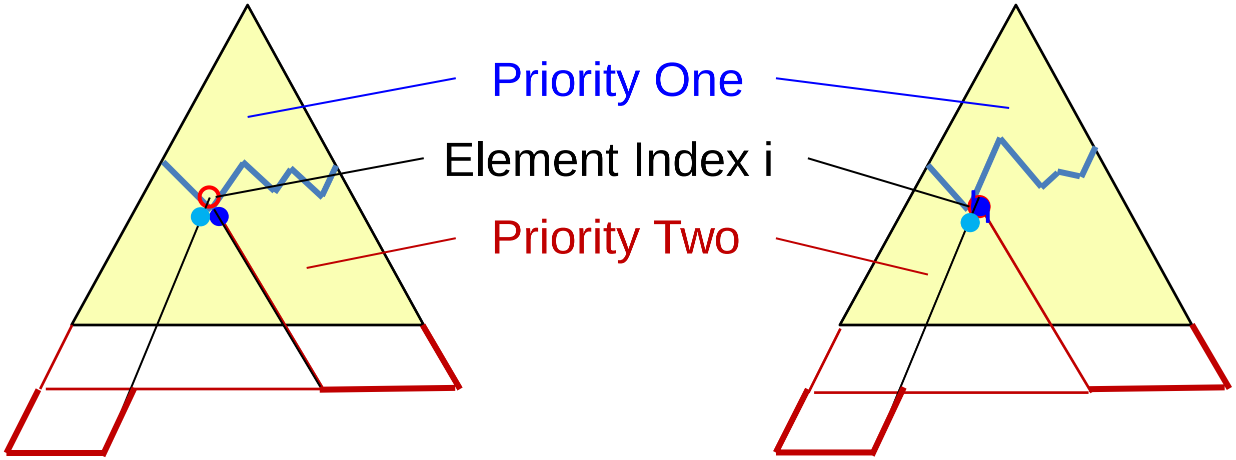



Push a <u>Different</u> Priority-One Value

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

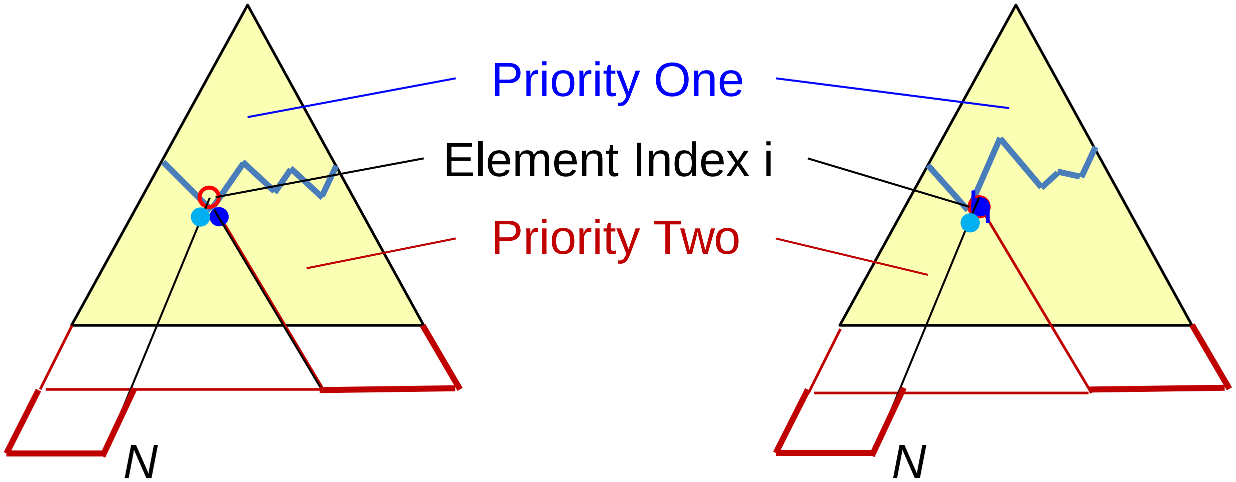



Push a Different Priority-One Value

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

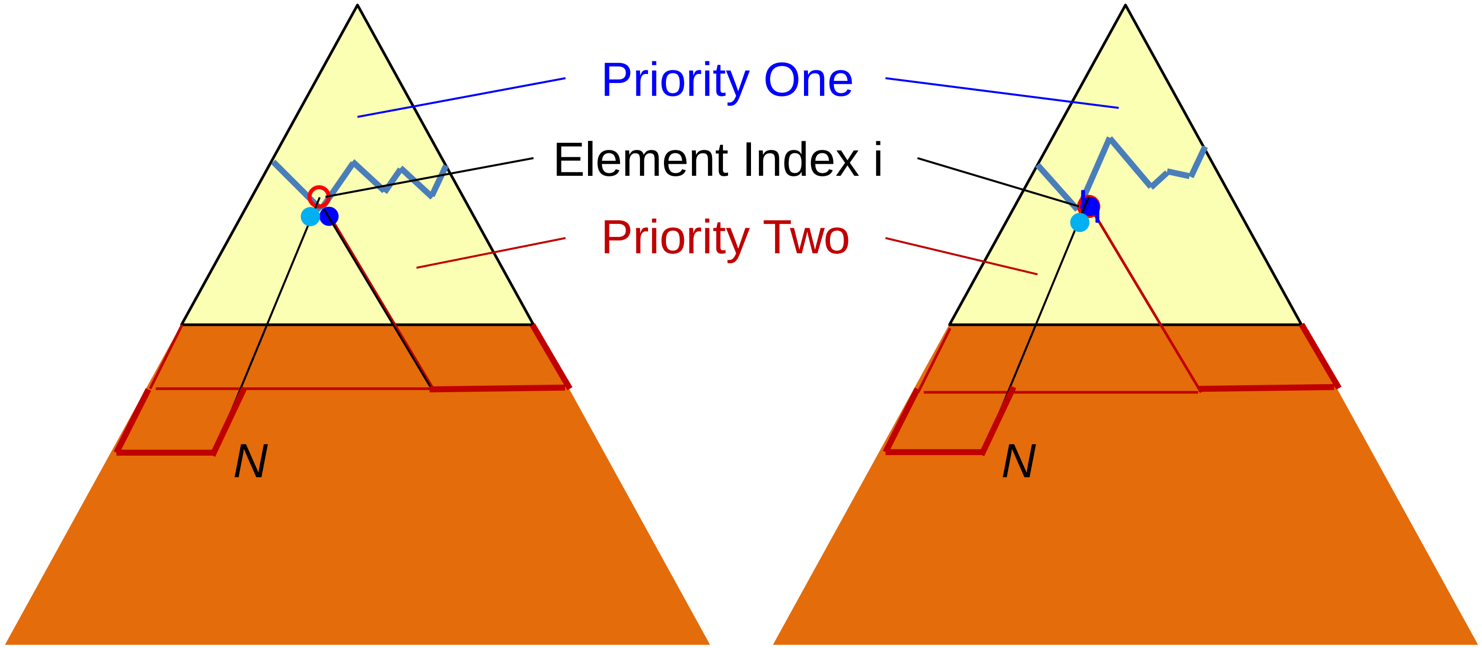



Push *N* Arbitrary Priority-Two Values

## Priority Queues

# Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

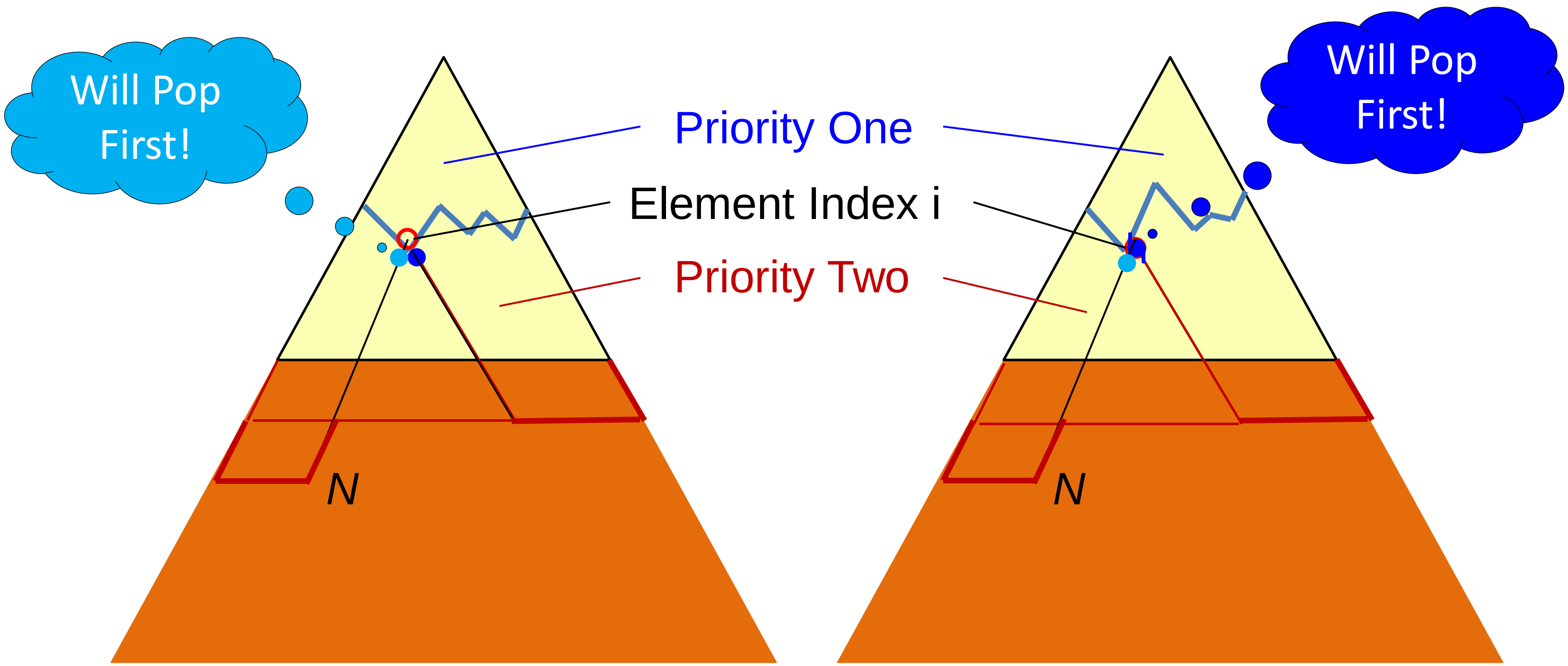



Push *N* Arbitrary Priority-Two Values

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

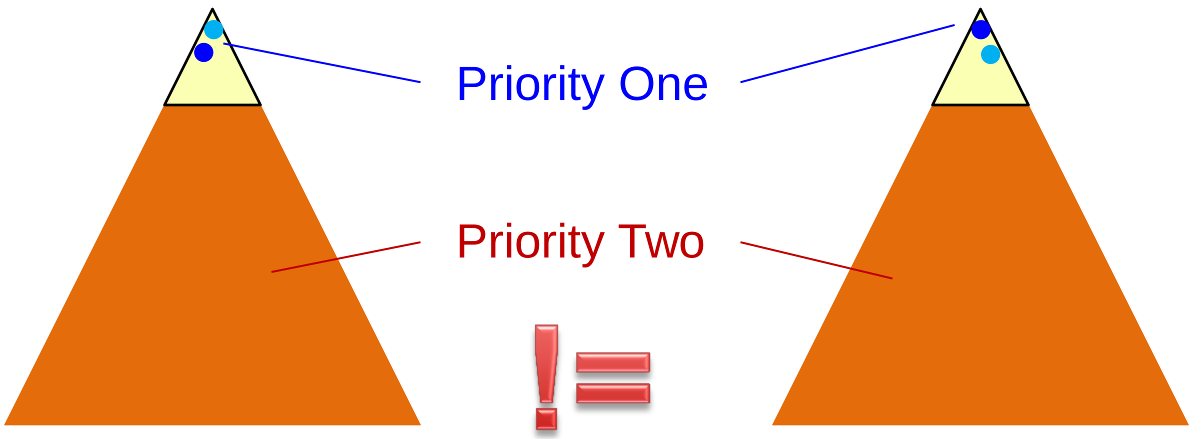



Pop *N* Elements

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?



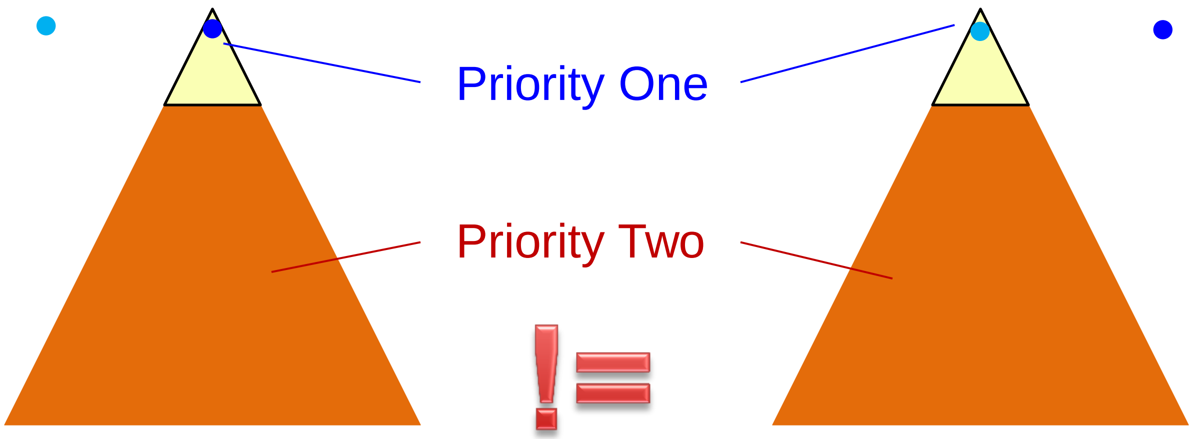

After almost N `pop` operations

**the tops are not the same!**

# Priority Queues

## **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?




After one more `pop` operation

**the element values are not the same!**

## Priority Queues

# **Should such a class be regular?**

Question: How *expensive* would `operator==` be to implement?

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

# YES IT SHOULD!

# Priority Queues

## Should such a class be regular?

Question: How *expensive* would `operator==` be to implement?

**YES IT SHOULD!**

**O[N] !!!**

# Priority Queues

# Discussion?

# Outline

1. Introduction and Background

    Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

    Most importantly, the ***Essential property of Value***

3. Two Important, Instructional Case Studies

    Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

    What must be remembered when designing value types

# Outline

1. Introduction and Background

    Components, Physical Design, and Class Categories

2. Understanding Value Semantics (and Syntax)

    Most importantly, the ***Essential property of Value***

3. Two Important, Instructional Case Studies

    Specifically, *Regular Expressions* and *Priority Queues*

4. Conclusion

    What must be remembered when designing value types

# What to Remember about VSTs

# What to Remember about VSTs

So what are the take-aways?

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.
- Ideally, each value type will have *regular* syntax.

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.
- Ideally, each value type will have *regular* syntax.
- Moreover, all operations on value types should follow proper *value semantics*:

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.
- Ideally, each value type will have *regular* syntax.
- Moreover, all operations on value types should follow proper *value semantics*:
  - Value derives only from autonomous object state, but not all object state need contribute to value.

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.
- Ideally, each value type will have *regular* syntax.
- Moreover, all operations on value types should follow proper *value semantics*:
  - Value derives only from autonomous object state, but not all object state need contribute to value.
  - Adhere to the *Essential Property of Value*.

# What to Remember about VSTs

## So what are the take-aways?

- Some types naturally represent a *value*.
- Ideally, each value type will have *regular* syntax.
- Moreover, all operations on value types should follow proper *value semantics*:
  - Value derives only from autonomous object state, but not all object state need contribute to value.
  - Adhere to the *Essential Property of Value*.
  - Behave as if each value has a canonical internal representation.

# What to Remember about VSTs

- Two objects of a given value-semantic type have the same value iff there does not exist a *distinguishing sequence* among all of its *salient* operations.

**Value is in a class's DNA**

# What to Remember about VSTs

The ***key*** take-away:

# What to Remember about VSTs

The ***key*** take-away:

What makes a value-type *proper* has essentially **nothing** to do with ***syntax***…

# What to Remember about VSTs

The ***<u>key</u>*** take-away:

What makes a value-type *proper* has essentially **<u>nothing</u>** to do with ***syntax***; it has **<u>everything</u>** to do with ***semantics***:

4. Conclusion

# What to Remember about VSTs

The ***key*** take-away:

What makes a value-type *proper* has essentially **nothing** to do with ***syntax***; it has **everything** to do with ***semantics***: A class that respects the ***Essential Property of Value*** is value-semantic…

# What to Remember about VSTs

The ***key*** take-away:

What makes a value-type *proper* has essentially **nothing** to do with ***syntax***; it has **everything** to do with ***semantics***: A class that respects the ***Essential Property of Value*** is value-semantic; **otherwise, it is not!**

# For More Information

- Find our open-source distribution at: **`http://www.openbloomberg.com/bde`**
- Moderator: **kpfleming@bloomberg.net**
- How to contribute? *See our site.*
- All comments and criticisms welcome...
- I can be reached at **jlakos@bloomberg.net**

# The End