Value Semantics

*It ain’t about the syntax!*

John Lakos

Thursday, April 23, 2015
Copyright Notice

© 2014 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.
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?
1. Introduction and Background

What’s the Problem?

Large-Scale C++ Software Design:

• Involves many subtle *logical* and *physical* aspects.
1. Introduction and Background

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

![Logical and Physical Design Diagram]
1. Introduction and Background

Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

Logical: Classes and Functions
What distinguishes *Logical* from *Physical* Design?

**Logical:** Classes and Functions

**Physical:** Files and Libraries
1. Introduction and Background

*Component: Uniform Physical Structure*

**A Component Is Physical**

```cpp
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
    // ...
}
//-- END OF FILE --

// component.h
// ...

//-- END OF FILE --

// component.cpp
#include <component.h>
// ...

//-- END OF FILE --
```
1. Introduction and Background

**Component: Uniform Physical Structure**

Implementation

```
// component.t.cpp
#include <component.h>
// ...
int main(...) {
  //...
}
//-- END OF FILE --

// component.h
// ...

//-- END OF FILE --

// component.cpp
#include <component.h>
// ...

//-- END OF FILE --
```

component.t.cpp

component.h

component.cpp

component
1. Introduction and Background

Component: Uniform Physical Structure
1. Introduction and Background

Component: Uniform Physical Structure

Test Driver
1. Introduction and Background

**Component: Uniform Physical Structure**

**The Fundamental Unit of Design**

```cpp
// component.t.cpp
#include <component.h>
// ...
int main(...) {
   // ...
}
//-- END OF FILE --

// component.cpp
#include <component.h>
// ...
//-- END OF FILE --

// component.h
// ...
//-- END OF FILE --

// component.h
#include <component.h>
// ...
//-- END OF FILE --

// component.cpp
#include <component.h>
// ...
//-- END OF FILE --
```
1. Introduction and Background

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*. 
1. Introduction and Background

Logical versus Physical Design

*Logical* content aggregated into a *Physical* hierarchy of *components*
1. Introduction and Background

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.
1. Introduction and Background

Implied Dependency

PointList

PointList_Link

Point

Polygon

Shape

Uses in the Interface

Uses in the Implementation

Uses in name only

Depends-On

Is-A
1. Introduction and Background

Implied Dependency

- PointList
- PointList_Link
- Polygon
- Point
- Shape

Diagram symbols:
- Uses-in-the-Interface
- Uses-in-the-Implementation
- Depends-On
- Uses in name only
- Is-A
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*. 
1. Introduction and Background

The Big Picture

- Type only
  - utility
  - meta-function
  - protocol
  - bteso::InetStreamSocketFactory
  - bslnf::IsFundamental
  - baetzo::Loader

- Mechanism
  - reference
type
  - guard/
proctor
  - factory
  - singleton
  - stateless
functor
  - btlsm::AlignmentUtil
  - bslmf::IsFundamental
  - baetzo::Loader

- Takes allocator?
  - yes
  - bslma::NewDeleteAllocator
  - bslma::DeallocatorGuard
  - bslma::DestructorProctor
  - container:
    - associative?
    - X ordered?
    - X unique?
    - X indexed?
    - general VST
    - bdet::Date
    - baetzo::LocalTimeValidity
    - attribute
    - complex constrained
    - composition
    - baet::LocalDateTime
    - baetzo::LocalTimePeriod
    - pure
    - unconstrained
    - simply constrained
    - externalizable
    - baet::LocalTimeDescriptor
    - baetzo::LocalTimeOptions
    - externalization
    - available from
    - bslox

- Is object
  - instantiable?
    - no
    - yes
    - utility
    - meta-function
    - protocol
    - bteso::InetStreamSocketFactory
    - bslnf::IsFundamental
    - baetzo::Loader

- Has "value"?
  - no
  - externalizable, no allocator
  - yes
  - btlm::NewDeleteAllocator
  - bslma::DeallocatorGuard
  - bslma::DestructorProctor
  - container:
    - associative?
    - X ordered?
    - X unique?
    - X indexed?
    - general VST
    - bdet::Date
    - baetzo::LocalTimeValidity
    - attribute
    - complex constrained
    - composition
    - baet::LocalDateTime
    - baetzo::LocalTimePeriod
    - pure
    - unconstrained
    - simply constrained
    - externalizable
    - baet::LocalTimeDescriptor
    - baetzo::LocalTimeOptions
    - externalization
    - available from
    - bslox

- Value
  - Semantic Type
  - yes
  - btlm::NewDeleteAllocator
  - bslma::DeallocatorGuard
  - bslma::DestructorProctor
  - container:
    - associative?
    - X ordered?
    - X unique?
    - X indexed?
    - general VST
    - bdet::Date
    - baetzo::LocalTimeValidity
    - attribute
    - complex constrained
    - composition
    - baet::LocalDateTime
    - baetzo::LocalTimePeriod
    - pure
    - unconstrained
    - simply constrained
    - externalizable
    - baet::LocalTimeDescriptor
    - baetzo::LocalTimeOptions
    - externalization
    - available from
    - bslox

- Externalizable?
  - no
  - externalizable, no allocator
  - yes
  - btlm::NewDeleteAllocator
  - bslma::DeallocatorGuard
  - bslma::DestructorProctor
  - container:
    - associative?
    - X ordered?
    - X unique?
    - X indexed?
    - general VST
    - bdet::Date
    - baetzo::LocalTimeValidity
    - attribute
    - complex constrained
    - composition
    - baet::LocalDateTime
    - baetzo::LocalTimePeriod
    - pure
    - unconstrained
    - simply constrained
    - externalizable
    - baet::LocalTimeDescriptor
    - baetzo::LocalTimeOptions
    - externalization
    - available from
    - bslox

- packed container
  - externalization
  - from bslox

- standard container
  - externalization
  - from bslox
1. Introduction and Background

The Big Picture

YOU ARE HERE
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
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

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.
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.
• Let’s begin with understanding some basic properties of 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.
• 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.
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.
• 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.
2. Understanding Value Semantics

True Story

• Date: Friday Morning, October 5th, 2007
• Place: LWG, Kona, Hawaii
• Defect: issue #684: Wording of Working Paper
2. Understanding Value Semantics

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” ?
What do we mean by “the same”?
2. Understanding Value Semantics

“The Same”

What do we mean by “the same”?

• The two objects are *identical*?
  – same address, same process, same time?
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.
2. Understanding Value Semantics

“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.)
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...
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?
2. Understanding Value Semantics

What exactly has to be “the Same”?

The discussion continued...

...some voiced suggestions:
2. Understanding Value Semantics

What exactly has to be “the Same”?

The discussion continued...

...some voiced suggestions:

• Whatever the copy constructor preserves.
2. Understanding Value Semantics

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”.
2. Understanding Value Semantics

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”.
2. Understanding Value Semantics

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!!”
2. Understanding Value Semantics

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”?
2. Understanding Value Semantics

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.
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?
2. Understanding Value Semantics

What does a *Copy Constructor* do?
2. Understanding Value Semantics

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”.
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

Same *Object*?
2. Understanding Value Semantics

**Same Object?**

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

assert(&a == &b); // ?
```
2. Understanding Value Semantics

Same *Object*?

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

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

assert(0 == b.size());

a.push_back(5.0);

assert(1 == b.size()); // ??
```
2. Understanding Value Semantics

**Same Object?**

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

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

assert(1 == b.size()); // ??
```
2. Understanding Value Semantics

**Same Object?**

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

assert(&a == &b); // ?
assert(0 == b.size());
a.push_back(5.0);
assert(1 == b.size()); // ??
```
2. Understanding Value Semantics

Same Object?

```cpp
std::vector<double> a, b(a);
assert(&a == &b); // ??
assert(0 == b.size());
a.push_back(5.0);
assert(1 == b.size()); // ??
```
2. Understanding Value Semantics

**Same Object?**

```cpp
std::vector<double> a, b(a);
assert(&a == &b); // ??
assert(0 == b.size());
a.push_back(5.0);
assert(1 == b.size()); // ??
```
2. Understanding Value Semantics

**Same State?**
2. Understanding Value Semantics

Same State?

class String {
    char *d_array_p;  // dynamic
    int d_capacity;
    int d_length;

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

public:
    String();
    String(const String& original);
    // ...
};

What happens if this address is copied?
2. Understanding Value Semantics

Same *Behavior*?
If we apply the same sequence of operations to both objects, the observable behavior will be the same:
2. Understanding Value Semantics

**Same Behavior?**

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

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

Same *Behavior*?

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

```cpp
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?
}```
2. Understanding Value Semantics

**Same Behavior?**

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

```cpp
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"; }
}
```
2. Understanding Value Semantics

Same *What*?
2. Understanding Value Semantics

Same *What*?

What should be “the same” after copy construction?
2. Understanding Value Semantics

Same *What*?

What should be “the same” after copy construction?

(It better be easy to understand.)
2. Understanding Value Semantics

Same *What?*

What should be “the same” after copy construction?

(It better be easy to understand.)

The two objects should *represent* the same *value*! 
2. Understanding Value Semantics

What do we mean by “value”?
2. Understanding Value Semantics

What do we mean by “value”?
2. Understanding Value Semantics

Mathematical Types
A mathematical type consists of

• A set of globally **unique values**
  – Each one describable independently of any particular representation.
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, 👉, 101 (binary), five, 🍀
2. Understanding Value Semantics

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:
    
    $5, \ 5, \ \text{V}, \ 101 \text{ (binary)}, \ \text{five}, \ \text{IIII}$

- A set of **operations** on those values
  - For example: $+, \ -, \ \times \ \ (3 + 2)$
2. Understanding Value Semantics

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:
    \[ 5, 5, \sqrt{}, 101{\text{ (binary)}}, \text{five}, \Box \]

• A set of operations on those values
  – For example: \( +, -, \times \) \( (3 + 2) \)
2. Understanding Value Semantics

C++ Type
2. Understanding Value Semantics

C++ Type

• A C++ type *may* represent (an approximation to) an abstract mathematical type:
2. Understanding Value Semantics

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`. 
2. Understanding Value Semantics

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.
2. Understanding Value Semantics

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.
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();
};
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();
};
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();
};
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() const;
    int month() const;
    int day() const;
};
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();
};

class Date {
    int d_serial;

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();
};

class Date {
  int d_serial;
  public:
    //
    int year();
    int month();
    int day();
};
So, what do we mean by “value”?

Salient Attributes

```cpp
int year();
int month();
int day();
```
2. Understanding Value Semantics

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.
2. Understanding Value Semantics

So, what do we mean by “value”?

```cpp
class Time {
    char d_hour;
    char d_minute;
    char d_second;
    short d_millisec;

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

```cpp
class Time {
    int d_mSeconds;

public:
    // ...
    int hour();
    int minute();
    int second();
    int millisecond();
};
```
2. Understanding Value Semantics

So, what do we mean by “value”?

class Time {
    // Internal Representation
    public:
    // ...
    int hour();
    int minute();
    int second();
    int millisecond();
};
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

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!
2. Understanding Value Semantics

So, what do we mean by “value”?

QUESTION:
What about integers and integers mod 5?
2. Understanding Value Semantics

So, what do we mean by “value”?

An “interpretation” of a subset of *instance* state.
2. Understanding Value Semantics

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*. 
2. Understanding Value Semantics

So, what do we mean by “value”?

class Point {
    short int d_x;
    short int d_y;

    public:
    // ...
    int x();
    int y();
};
2. Understanding Value Semantics

So, what do we mean by “value”?

class Point {
    // Internal Representation

    public:
    // ...
    int x();
    int y();

};
2. Understanding Value Semantics

So, what do we mean by “value”?

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

class Box {
    Point d_topLeft;
    Point d_botRight;
    public:
        // ...
        Point origin();
        int length();
        int width();
};
```
2. Understanding Value Semantics

So, what do we mean by “value”?

```cpp
class Point {
    public:
        // ...
        int x();
        int y();
};

class Box {
    public:
        // ...
        Point origin();
        int length();
        int width();
};
```
2. Understanding Value Semantics

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
2. Understanding Value Semantics

What are “Salient Attributes”?
2. Understanding Value Semantics

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);  // ...
};
2. Understanding Value Semantics

What are “Salient Attributes”? 

Consider `std::vector<int>`:

What are its *salient attributes*?
2. Understanding Value Semantics

What are “Salient Attributes”? 

Consider `std::vector<int>`:

What are its *salient attributes*?

1. The number of elements: `size()`.
2. Understanding Value Semantics

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.
2. Understanding Value Semantics

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()`?
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

What are “Salient Attributes”? 

Salient Attributes:
2. Understanding Value Semantics

What are “Salient Attributes”?

**Salient Attributes:**

1. Are a part of logical design.
2. Understanding Value Semantics

What are “Salient Attributes”?

Salient Attributes:

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


2. Understanding Value Semantics

What are “Salient Attributes”? 

Salient Attributes:
1. Are a part of logical design.
2. *Should* be “natural” & “intuitive”
3. Must be documented *explicitly*!
Why is value important?
2. Understanding Value Semantics

Why are unique values important?
2. Understanding Value Semantics

Why are unique values important?

IPC

Inter-Process Communication
2. Understanding Value Semantics

Why are unique values important?

Abstract \textit{date} Type \hspace{2cm} \texttt{C++ Date Class}
2. Understanding Value Semantics

Why are unique values important?

Abstract *date* Type
Has an infinite set of valid *date* values.

C++ *Date* Class
2. Understanding Value Semantics

Why are **unique values** important?

Abstract *date* Type

Has an infinite set of valid *date* values.

C++ *Date* Class

Globally Unique Values
2. Understanding Value Semantics

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
2. Understanding Value Semantics

Why are unique values important?

Abstract *date* Type

Has an infinite set of valid *date* values.

- 1000000 B.C.
- 1969-07-16
- 1959-03-08
- 1941-12-07
- 2008-04-03
- 1999-12-31
- 1917-12-10
- 1949-02-05
- 1994-08-14
- 2001-09-11
- 1999-12-31
- 1776-07-04
- 1000000 A.D.

C++ *Date* Class

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

Globally Unique Values
2. Understanding Value Semantics

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
2. Understanding Value Semantics

Why are unique values important?

Globally Unique Values
2. Understanding Value Semantics

Why are **unique values** important?

Database

Globally Unique Values

C++

- 1000000 B.C.
- 1959-03-08
- 99999-12-31
- 1994-08-14
- 1999-12-31
- 1000000 A.D.
- 1969-07-16
- 1941-12-07
- 1917-12-10
- 2008-04-03
- 1919-02-05
- 2001-09-11
- 1776-07-04

Date
2. Understanding Value Semantics

Why are **unique values** important?

Globally Unique Values

Database

- 1000000 B.C.
- 1959-03-08
- 99999-12-31
- 1994-08-14
- 1999-12-31
- 1000000 A.D.
- 1969-07-16
- 1941-12-07
- 2008-04-03
- 1917-12-10
- 1919-02-05
- 2001-09-11
- 1776-07-04

Date

C++
2. Understanding Value Semantics

Why are **unique values** important?
2. Understanding Value Semantics

Why are **unique values** important?

Java

Database

Globally Unique Values

C++
2. Understanding Value Semantics

Why are unique values important?

Globally Unique Values

Java

Date

Database

C++
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

Does **state** *always* imply a "**value**"?
2. Understanding Value Semantics

Does **state** *always* imply a “**value**”?
2. Understanding Value Semantics

Does state *always* imply a “value”?
2. Understanding Value Semantics

Does state *always* imply a “value”?

What is its state?
2. Understanding Value Semantics

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

What is its state? OFF
2. Understanding Value Semantics

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

What is its state?
2. Understanding Value Semantics

Does \textbf{state} \textit{always} imply a “\textit{value}”? 

What is its state? ON
2. Understanding Value Semantics

Does state \textit{always} imply a “value”? 

What is its state? ON
What is its value?
2. Understanding Value Semantics

Does state always imply a “value”? 

What is its state? ON
What is its value? ?
2. Understanding Value Semantics

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

What is its state?  **ON**

What is its value?  1  ?
2. Understanding Value Semantics

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

What is its state? **ON**

What is its value? **false**?
2. Understanding Value Semantics

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

What is its state?  **ON**

What is its value?  **£5.00**?
2. Understanding Value Semantics

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

What is its state?  **ON**

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

Cheap at half the price!
2. Understanding Value Semantics

Does state always imply a “value”?

What is its state? ON
What is its value? ?

Any notion of “value” here would be artificial!
2. Understanding Value Semantics

Does state always imply a “value”?

Not every stateful object has an obvious value.
2. Understanding Value Semantics

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
2. Understanding Value Semantics

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

What would copy construction even **mean** here?
2. Understanding Value Semantics

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

What would copy construction even mean here?

We could invent some notion of value, but to what end??
2. Understanding Value Semantics

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
2. Understanding Value Semantics

Does \textbf{state} \textit{always} imply a “\textit{value}”? \\

\textbf{QUESTION:} \\
Suppose we have a thread-safe queue used for inter-task communication: Is it a value type?
2. Understanding Value Semantics

Does `state always` imply a “value”?

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?
2. Understanding Value Semantics

Does state always imply a “value”?

We refer to stateful objects that do not represent a value as “Mechanisms”.

- TCP/IP Socket
- Thread Pool
- Condition Variable
- Mutex
- Reader/Writer Lock
- Scoped Guard
- Base64 Encoder
- Expression Evaluator
- Language Parser
- Event Logger
- Object Persistor
- Widget Factory
2. Understanding Value Semantics

The Big Picture

YOU ARE HERE
2. Understanding Value Semantics

Categorizing Object Types

MyObjectType
Categorizing Object Types

The first question: “Does it have state?”
2. Understanding Value Semantics

Categorizing Object Types

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

2. Understanding Value Semantics

Categorizing Object Types

- DateUtil
- std::less<T>
- IsConvertible<U,V>

- Stateless Object
- Stateful Object
- Object
The first question: “Does it have state?”

```cpp
define 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.

    // ...
}
```
The first question: “Does it have state?”

```cpp
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.

    // ...
};
```

Utilities are an important class category!
2. Understanding Value Semantics

The Big Picture

Common Category

YOU ARE HERE
2. Understanding Value Semantics

Categorizing Object Types

The second question: “Does it have value?”
2. Understanding Value Semantics

Categorizing Object Types

The second question: “Does it have value?”
2. Understanding Value Semantics

Categorizing Object Types

The second question: “Does it have value?”
The second question: “Does it have value?”

No

- **Stateless Object**
- **Stateful Object**

Mechanism

Value Type
2. Understanding Value Semantics

Top-Level Categorizations

- **Start here**

- **Is object-instantiable?**
  - no
  - yes
  - Takes allocator?

- **Has “value”?**
  - no
  - yes

- **Value-Semantic Type**

- **Type only**

- **Mechanism**

[Diagram showing the flow of decisions and categorizations related to value semantics and instantiability.]
2. Understanding Value Semantics

Top-Level Categorizations

- Is object-instantiable?
  - yes: Takes allocator?
  - no: Type only

- Has “value”?
  - yes: Value-Semantic Type
  - no: Mechanism
2. Understanding Value Semantics

Top-Level Categorizations

- Is object-instantiable?
  - no: Type only
  - yes: Takes allocator?
    - no: Has “value”?
      - no: Mechanism
      - yes: Value-Semantic Type
    - yes: Takes allocator?
2. Understanding Value Semantics

The Big Picture
2. Understanding Value Semantics

The Big Picture

Common Category

- utility
- meta-function
- protocol

Type only

- bsls::AlignmentUtil
- bslmf::IsFundamental
- baetzo::LocalTimeDescriptor

Common Category

- Takes allocator?
- Has "value"?

Mechanism

- reference semantic type
- guard/proctor
- factory
- singleton

- bdem::ElemRef
- balma::DeallocatorGuard
- balma::DestructorProctor
- bteso::InetStreamSocketFactory
- bslma::NewDeleteAllocator

- btl::less
- btl::vector
- btl::BitArray

- packed container
- standard container

- externalization available from bslx

Common Category

- Value-Semantic Type
- container:
  - associative?
  - X ordered?
  - X unique?
  - X indexed?

- unconstrained
- constrained

Pure abstract interface
2. Understanding Value Semantics

The Big Picture

QUESTION:
What does it mean for two abstract types to compare equal?
QUESTION:
What does it mean for two abstract types to compare equal?
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

Value-Semantic Properties

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

• Default construction:  T a, b;    assert(a == b);
A value-semantic type $T$ defines the following:

- Default construction: $T \ a, \ b; \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ assert(a == b);$
A value-semantic type $T$ defines the following:

- Default construction: $T a, b; \quad \text{assert}(a == b);$

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

However “zero” initialization

\[
\text{assert}(T() == T()); \\
\text{Is true}
\]
2. Understanding Value Semantics

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);`
2. Understanding Value Semantics

Value-Semantic Properties

A value-semantic type \( T \) defines the following:

- Default construction: \( T \ a, b; \ \text{assert}(a == b); \)
- Copy construction: \( T \ a, b(a); \ \text{assert}(a == b); \)
- Destruction: (\textit{Release all resources.})
2. Understanding Value Semantics

Value-Semantic Properties

A value-semantic type T defines the following:

• Default construction: \[ T \ a, \ b; \quad \text{assert}(a == b); \]

• Copy construction: \[ T \ a, \ b(a); \quad \text{assert}(a == b); \]

• Destruction: \[ (Release \ all \ resources.) \]

• Copy assignment: \[ a = b; \quad \text{assert}(a == b); \]
2. Understanding Value Semantics

Value-Semantic Properties

A value-semantic type T defines the following:

• Default construction: \( T \ a, \ b; \ \text{assert}(a == b); \)

• Copy construction: \( T \ a, b(a); \ \text{assert}(a == b); \)

• Destruction: \( (Release \ all \ resources. ) \)

• Copy assignment: \( a = b; \ \text{assert}(a == b); \)

• Swap (if well-formed): \( T \ a(\alpha), b(\beta); \ \text{swap}(a, b); \)
  \( \text{assert}(\beta == a); \)
  \( \text{assert}(\alpha == b); \)
2. Understanding Value Semantics

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): $T a(\alpha), b(\beta)$; $swap(a, b)$;
  
  $assert(\beta == a)$;
  
  $assert(\alpha == b)$;
operator==(T, T) describes what’s called an equivalence relation:

1. \( a == a \) (reflexive)
2. \( a == b \Leftrightarrow b == a \) (symmetric)
3. \( a == b \land b == c \Rightarrow a == c \) (transitive)
operator==(T, T) describes what’s called an equivalence relation:

1. \( a == a \)  
   (reflexive)

2. \( a == b \Leftrightarrow b == a \)  
   (symmetric)

3. \( a == b \land b == c \Rightarrow a == c \)  
   (transitive)

\[ \neg (a == b) \Leftrightarrow a != b \]
operator==(T, T) describes what’s called an equivalence relation:

1. \( a == a \) (reflexive)
2. \( a == b \iff b == a \) (symmetric)
3. \( a == b \land b == c \Rightarrow a == c \) (transitive)

- \( ! (a == b) \iff a != b \)
- \( a == d \) (compiles) \( \iff d == a \) (compiles)

(Note that \( d \) is not of the same type as \( a \).)
operator== (T, T) describes what's called an equivalence relation:

1. $a == a$ (reflexive)
2. $a == b \iff b == a$ (symmetric)
3. $a == b \land b == c \implies a == c$ (transitive)

$\neg (a == b) \iff a != b$ (compiles)

$\neg a == d$ (compiles) $\iff d == a$ (compiles)

(Note that $d$ is not of the same type as $a$.)
2. Understanding Value Semantics

Value-Semantic Properties

Member operator==

class T {
    // ...
    public:
    // ...
    bool operator==(const T& rhs) const;
    // ...
};
2. Understanding Value Semantics

Value-Semantic Properties

\[ \text{Member} \ \text{operator}== \]

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

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

Value-Semantic Properties

\textit{Member} \texttt{operator==}
2. Understanding Value Semantics

Value-Semantic Properties

Member operator==

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

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

```cpp
void f(const T& a, const D& d)
{
    if (a == d) {/* ... */}
    if (d == a) {/* ... */}
}
```
2. Understanding Value Semantics

Value-Semantic Properties

Free operator==

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

```cpp
class T {
   public:
      // ...
};

void f(const T& a, const D& d)
{
   if (a == d) { /* ... */ }
}
```
2. Understanding Value Semantics

Value-Semantic Properties

Free `operator==`

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

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

(proper)

```cpp
void f(const T& a, const D& d)
{
    if (a == d) { /* ... */ }
    if (d == a) { /* ... */ }
}
```
2. Understanding Value Semantics

Value-Semantic Properties

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

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

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

2. Understanding Value Semantics

Value-Semantic Properties
2. Understanding Value Semantics

Value-Semantic Properties

```cpp
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);
    // ...
};
```
2. Understanding Value Semantics

Value-Semantic Properties

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

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

Value-Semantic Properties

```cpp
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);
```
2. Understanding Value Semantics

Value-Semantic Properties

```cpp
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);
```
### 2. Understanding Value Semantics

#### Value-Semantic Properties

```cpp
class Str {
    Member Operator==
    public:
        Str(const char *other);
        bool operator==(const char *rhs) const;
    };

bool operator==(const char *lhs, const Str &rhs);
bool operator==(const char *lhs, const Str &rhs);

void f(const Foo &foo, const Bar &bar)
{
    if (bar == foo) { /* … */ }
    if (foo == bar) { /* … */ }
}
```
2. Understanding Value Semantics

Value-Semantic Properties

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

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

```cpp
class Foo {
    // ...
    public:
    // ...
};

class Bar {
    // ...
    public:
    // ...
};
```

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

Free Operator==
2. Understanding Value Semantics

Value-Semantic Properties

```cpp
class Str {
    public:
        Str(const char *other);
    // …
};

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

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

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

```cpp
void f(const Foo &foo, const Bar &bar)
{
    if (bar == foo) { /* … */ }
    if (foo == bar) { /* … */ }
}
```
The operator == should **ALWAYS** be free!

### 2. Understanding Value Semantics

#### Value-Semantic Properties

- **Equality** (`==`)
- **Relational** (`<`, `<=`, `>`, `=>`)
- **Arithmetic** (`+`, `-`, `*`, `/`, `%`)
- **Logical** (`|`, `&`, `^`, `<<`, `>>`)
- **Assignment** (`+=`, `-=` , `*=`, `/=`, `%=`)
2. Understanding Value Semantics

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()`.*
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()`.*
2. Understanding Value Semantics

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()`.*
2. Understanding Value Semantics

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()`.
2. Understanding Value Semantics

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()`.*
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()`.*
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()`.*
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?
2. Understanding Value Semantics

Where is “Value” Defined?
The *salient attributes* of a type $\mathcal{T}$ are the documented set of named attributes whose respective values for a given instance of $\mathcal{T}$ ...
2. Understanding Value Semantics

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 \( T \).
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.*
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.
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.
2. Understanding Value Semantics

Where is “Value” Defined?

Copy Constructor?

• By def., all salient attributes must be copied.
By def., all salient attributes must be copied.

What about “non-salient” attributes?

– E.g., capacity()
2. Understanding Value Semantics

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.

More on this later...
2. Understanding Value Semantics

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.”
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!
2. Understanding Value Semantics

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*.
2. Understanding Value Semantics

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*. 
2. Understanding Value Semantics

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.
2. Understanding Value Semantics

Where is “Value” Defined?

operator==

The associated, homogeneous (free) operator== for a type T
2. Understanding Value Semantics

Where is "Value" Defined?

operator==

The associated, homogeneous (free) operator== for a type \( \mathbb{T} \)

1. Provides an \textit{operational definition} of what it means for two objects of type \( \mathbb{T} \) to have "the same" value.
2. Understanding Value Semantics

Where is “Value” Defined?

operator==

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

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.
2. Understanding Value Semantics

Value-Semantic Properties

Value-semantic objects share many properties.
Value-semantic objects share many properties.

- Each of these properties is objectively verifiable, irrespective of the intended application domain.
2. Understanding Value Semantics

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.
2. Understanding Value Semantics

What should be copied?

Should attributes that are orthogonal to value be copied?

Orthogonal to value!

Orthogonal to value!

NMOS

0 1 0 0 1 0 0 1

CMOS

1 1 1 1 0 0 1 1
2. Understanding Value Semantics

What should be copied?

Should attributes that are **orthogonal** to value be copied?

![Diagram showing NMOS and CMOS with binary values 01001001 and 11110011, respectively.](image-url)
2. Understanding Value Semantics

What should be copied?

Should attributes that are **orthogonal** to value be copied?

![Diagram showing comparison between NMOS and CMOS attributes](image-url)
2. Understanding Value Semantics

What should be copied?

Should attributes that are orthogonal to value be copied?

NMOS: 0 1 0 0 1 0 0 1
CMOS: 1 1 1 1 0 0 1 1

assignment
Should attributes that are orthogonal to value be copied?

2. Understanding Value Semantics

What should be copied?

NMOS

CMOS

0 1 0 0 1 0 0 1

1 1 1 1 0 0 1 1

assignment
2. Understanding Value Semantics

What should be copied?

Should attributes that are orthogonal to value be copied?

No Way!
2. Understanding Value Semantics

What should be copied?

V-TABLE POINTER?

ALLOCATOR?
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

As it turns out...

Choosing salient attributes appropriately will affect our ability to test thoroughly.
If $\text{T}$ is a value-semantic type, $a, b,$ and $c$ are objects of type $\text{T}$, and $d$ is an object of some other type $\text{D}$, then
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

> $a == b \iff a$ and $b$ have the same value

(assuming an associated operator $==$ exists).
2. Understanding Value Semantics

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

\[ a == b \iff a \text{ and } b \text{ have the same value} \]

(assuming an associated operator $==$ exists).
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 

\[
\neg a == b \iff a \text{ and } b \text{ have the same value }
\]

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

$$a == b \iff a \text{ and } b \text{ have the same value}$$

(assuming an associated operator $==$ exists).

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

- $a == b \iff a$ and $b$ have the same value (assuming an associated operator $==$ exists).

- The value of $a$ is independent of any external object or state; any change to $a$ must be accomplished via $a$’s (public) interface.
2. Understanding Value Semantics

Value-Semantic Properties

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

```cpp
class ElemRef
{
    Record *d_record_p;
    int elementIndex;

public:
// …
};
```
2. Understanding Value Semantics

Value-Semantic Properties

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

```cpp
class ElemPtr {
    Record *d_record_p;
    int d_elementIndex;
    public:
        // ...
    
};
```
Suppose a “value-semantic” object refers to another autonomous object in memory:

class ElemPtr {
    Record *d_record_p;
    int d_elementIndex;

    public:
        // ...

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

```cpp
class ElemPtr {
    Record *d_record_p;
    int    d_elementIndex;

public:
    // ...
};
```
2. Understanding Value Semantics

Value-Semantic Properties
2. Understanding Value Semantics

Value-Semantic Properties

Process Memory

0x002c5f20:

0x74a1254c:

0x74a1254c:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ElemPtr

Record
2. Understanding Value Semantics

Value-Semantic Properties

Process Memory

Record

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ElemPtr

0x74a1254c:

0x002c5f20:
2. Understanding Value Semantics

Value-Semantic Properties

```cpp
bool operator==(const ElemPtr& lhs,
               const ElemPtr& rhs);
```
bool operator==(const ElemPtr& lhs, const ElemPtr& rhs);
// Two 'ElemPtr' objects have the same value if they ...
2. Understanding Value Semantics

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) ...
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.
2. Understanding Value Semantics

Value-Semantic Properties

- Record objA
  - ElemPtr obj1
  - ElemPtr obj2
- Record objB
  - ElemPtr obj3
  - ElemPtr obj4
2. Understanding Value Semantics

Value-Semantic Properties

- Record objA
  - ElemPtr obj1
    - 0
    - 2
  - ElemPtr obj2
    - 0
    - 3

- Record objB
  - ElemPtr obj3
    - 0
    - 3
  - ElemPtr obj4
    - 0
    - 3
2. Understanding Value Semantics

Value-Semantic Properties

Record objA

ElemPtr obj1
0
2

ElemPtr obj2
0
3

Record objB

ElemPtr obj3
0
3

ElemPtr obj4
0
3
2. Understanding Value Semantics

Value-Semantic Properties
2. Understanding Value Semantics

Value-Semantic Properties
2. Understanding Value Semantics

Value-Semantic Properties
2. Understanding Value Semantics

Value-Semantic Properties

Record objA

Record objB

ElemPtr obj1

ElemPtr obj2

ElemPtr obj3

ElemPtr obj4

0 2

0 3

0

3

3

2. Understanding Value Semantics
2. Understanding Value Semantics

Value-Semantic Properties

Record objA == Record objB

ElemPtr obj1

ElemPtr obj2

ElemPtr obj3

ElemPtr obj4

0

2

3

3

0

0

3

271
2. Understanding Value Semantics

Value-Semantic Properties

In-Core Value Semantics

Record objA

ElemPtr obj2

ElemPtr obj3

ElemPtr obj4

ElemPtr obj1

0

0

2

3

272
2. Understanding Value Semantics

Value-Semantic Properties

I.E., NOT FULL VALUE SEMANTICS

Element obj2

Element obj3
2. Understanding Value Semantics

Value-Semantic Properties

In-Core Value Semantics, While Important, Is Not 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.
2. Understanding 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**.
A C++ type that “properly” represents (a subset of) the values of an abstract “mathematical” type is said to have value semantics.
Recall that two distinct objects \( a \) and \( b \) of type \( T \) that have the same value might not exhibit “the same” observable behavior.
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.
2. Understanding 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.*

**H owever**
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
Recall that two \textit{distinct} objects \texttt{a} and \texttt{b} of type T that have \textit{the same value} might \textbf{not} exhibit "the same" \textit{observable behavior}.

\textbf{However}

1. If \texttt{a} and \texttt{b} initially have the \underline{same value}, and
2. the \underline{same operation} is applied to each object, then
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
3. (absent any exceptions or undefined behavior)
2. Understanding 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.

However

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!
Note 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
3. (absent any exceptions or undefined behavior)
4. both objects will again have the same value!

2. Understanding Value Semantics

Value-Semantic Properties

There is a lot more to this story!

Deciding what is (not) salient is surprisingly important.

Subtle Essential Property of Value
That is...

```c
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);
    ...
}
```

2. Understanding Value Semantics

Value-Semantic Properties

**SUBTLE ESSENTIAL PROPERTY OF VALUE**
That is...

```c
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);
    // ...
}
```

Note that this is **not** a test case, but rather a requirements specification.
2. Understanding Value Semantics

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);
    // ...
};
2. Understanding Value Semantics

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:

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

What if the two sets were constructed differently.
2. Understanding Value Semantics

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:

```cpp
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”?
2. Understanding Value Semantics

Value-Semantic Properties

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

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!
2. Understanding Value Semantics

**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 operation* is applied to each object, then
3. (absent any exceptions or undefined behavior)
4. both objects will *again* have the *same value*!
QUESTION: What makes two unordered containers represent the same value?

Think about a bag of Halloween candy.
2. Understanding Value Semantics

Value-Semantic Properties

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

```c
int x = 5;    int y = 5;    assert(x == y);
```

```c
x *= x;    y *= y;    assert(x == y);
```

```c
x *= x;    y *= y;    assert(x == y);
```

```c
x *= x;    y *= y;    assert(x == y);
```

```c
x *= x;    y *= y;    assert(x == y);
```

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

```plaintext
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?
2. Understanding Value Semantics

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);
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

What about

numerator() / denominator() as the salient attribute?
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What about \texttt{numerator()}/\texttt{denominator()} as the salient attribute?

```cpp
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'.
```
Selecting Salient Attributes

Value-Semantic Properties

What about \( \text{numerator}() / \text{denominator}() \) as the salient attribute?

\[
\frac{1}{2} == \frac{2}{4} ?
\]
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What about

\texttt{numerator() / denominator()}

\textbf{as the salient attribute?}

\[
\frac{1}{0} \neq \frac{2}{0}
\]
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What about `numerator() / denominator()` as the salient attribute?

\[
\frac{1}{2} \quad == \quad \frac{-1}{-2}
\]
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What about

\[ \frac{\text{numerator()}}{\text{denominator()}} \]

as the salient attribute?

\[
\frac{1}{2} \quad == \quad \frac{100}{200}
\]
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What about \( \text{numerator}() / \text{denominator}() \) as the salient attribute?

\[
\begin{pmatrix}
\frac{1}{2}
\end{pmatrix}^{10} \text{ vs. } \begin{pmatrix}
\frac{100}{200}
\end{pmatrix}^{10}
\]

?
Value-Semantic Properties

Selecting Salient Attributes

VOLATES SUBTLE ESSENTIAL PROPERTY OF VALUE
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

If you choose to make

\[ \text{numerator}() / \text{denominator}() \]

a salient attribute

(\text{probably a bad idea})

then do not expose numerator and denominator as separate attributes...
If you choose to make `numerator()/denominator()` a salient attribute (probably a bad idea) then do not expose numerator and denominator as separate attributes...

...or maintain them in “canonical form” (which may be computationally expensive).
2. Understanding Value Semantics

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?
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

Graphs
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Graph

Node

Edge

Cyclic Physical Dependency?
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Graph

Node

Edge

Levelization Technique: Opaque Pointers
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Graph Levelization Technique: **Dumb Data**

- Graph
- Node
- Edge

graph
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Simpler Design: *No Explicit Edge Object*
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Yet Simpler Design: No Explicit NodeIterator
class Graph {
    // ...

public:
    // ...
    int numNodes() const;
    const Node& node(int index) const;
};
// ...

2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

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

2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Really should be declared const but there’s no room!
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

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

2. Understanding Value Semantics

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);
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 ...???
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What are the salient attributes of \textit{Graph}?
What are the salient attributes of **Graph**?

- Number of nodes.
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

What are the salient attributes of Graph?

- Number of nodes.
- Number of edges.
What are the salient attributes of \textit{Graph}?

- Number of nodes.
- Number of edges.
- Number of nodes adjacent to each node.
What are the salient attributes of Graph?

• Number of nodes.
• Number of edges.
• Number of nodes adjacent to each node.
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.
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.
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

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

2. Understanding Value Semantics

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'.
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

class Node {
    // ...

public:
    // ...
    int nodeIndex() const;
    int numAdjacentNodes() const;
    Node& adjacentNode(int index) const;
};
2. Understanding Value Semantics

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

<table>
<thead>
<tr>
<th>Unordered Edges</th>
<th>Ordered Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 4 1 3</td>
<td>0: 1 3 4</td>
</tr>
<tr>
<td>1: 3 2</td>
<td>1: 2 3</td>
</tr>
<tr>
<td>2: 0 4</td>
<td>2: 0 4</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
<tr>
<td>4: 3</td>
<td>4: 3</td>
</tr>
</tbody>
</table>

2. Understanding Value Semantics

- **Unordered Edges**
  - Node 0 connected to 4, 1, 3
  - Node 1 connected to 3, 2
  - Node 2 connected to 0, 4
  - Node 3 has no connections
  - Node 4 connected to 3

- **Ordered Edges**
  - Node 0 connected to 1, 3, 4
  - Node 1 connected to 2, 3
  - Node 2 connected to 0, 4
  - Node 3 has no connections
  - Node 4 connected to 3

![Diagram](image-url)
2. Understanding Value Semantics

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

Node

Edge

O[operator==]
2. Understanding Value Semantics

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

\[ O[N + E^2] \]

\[ O[\text{operator}==] \]
2. Understanding Value Semantics

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

$O[N + E^2]$
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Unordered Edges

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 4 1 3</td>
<td>2: 0 4</td>
</tr>
<tr>
<td>1: 3 2</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
</tr>
<tr>
<td>4: 3</td>
<td></td>
</tr>
</tbody>
</table>

\[O[N + E^2]\]

Ordered Edges

<table>
<thead>
<tr>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 1 3 4</td>
<td>2: 0 4</td>
</tr>
<tr>
<td>1: 2 3</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
</tr>
<tr>
<td>4: 3</td>
<td></td>
</tr>
</tbody>
</table>

\[O[N + E]\]

\[O\text{[operator==]}\]

Note that we could make it \[O[N + E*\text{Log}(E)]\].
2. Understanding Value Semantics

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

O[N + E^2]

O[operator==]

O[N + E]

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

Value-Semantic Properties

Observation
2. Understanding Value Semantics

Value-Semantic Properties

**Observation**

Value Syntax: *Not all or nothing!*
2. Understanding Value Semantics

**Value-Semantic Properties**

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,
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==`.
2. Understanding Value Semantics

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!**
2. Understanding Value Semantics

Value-Semantic Properties

Observation

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, this was due to value-semantic concerns!
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What are the salient attributes of Graph?
What are the salient attributes of Graph?

✓ Number of nodes.
What are the salient attributes of \textit{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 Not adjacent-node (i.e., edge) order.
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)
2. Understanding Value Semantics

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'.
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Node Index

3

1 2 3 4
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Node Index

1
2
3
4

0
1
2
3

4
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Node Index

Node Index 3
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Node Index

Should `operator==` mean *isomorphic*?
2. Understanding Value Semantics

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>
<thead>
<tr>
<th>Graph $G$</th>
<th>Graph $H$</th>
<th>An isomorphism between $G$ and $H$</th>
</tr>
</thead>
</table>
| ![Graph G](image1) | ![Graph H](image2) | $f(a) = 1$  
$f(b) = 6$  
$f(c) = 8$  
$f(d) = 3$  
$f(g) = 5$  
$f(h) = 2$  
$f(i) = 4$  
$f(j) = 7$ |

*http://en.wikipedia.org/wiki/Graph_isomorphism*
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

How hard is it to determine

\textit{Graph Isomorphism}?
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

How hard is it to determine \textit{Graph Isomorphism}? Is known to be in NP and CO-NP.
How hard is it to determine \textit{Graph Isomorphism}?

\textit{Is} known to be in NP \textit{and} CO-NP.  
\textit{Not} known to be NP Complete.
How hard is it to determine \textit{Graph Isomorphism}?

\textbf{Is known to be in NP \textit{and} CO-NP.}

\textbf{Not known to be NP Complete.}

\textbf{Not known to be in P (Polynomial time).}
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).
2. Understanding Value Semantics

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'.

2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

```cpp
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'.
```
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

```cpp
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'.
```
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

What are the salient attributes of Graph?

✓ Number of nodes.
✓ Specific nodes adjacent to each node.
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.
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

Or else we **Must Omit Equality Comparison Operators for this Class!**

- Numbering of the nodes.
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.

Or else we MUST 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?
2. Understanding Value Semantics

Discussion

Why would we ever omit valid value syntax when there...
2. Understanding Value Semantics

Discussion

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

When Doing So Is “Off Message!”
2. Understanding Value Semantics

Value-Semantic Properties
Selecting Salient Attributes

(Summary So Far)
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

(Summary So Far)

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

- Fractions may be equivalent, but not the same.
When selecting salient attributes, avoid subjective (domain-specific) interpretation:

- Fractions may be equivalent, but not the same.
- Graphs may be isomorphic, yet distinct.
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
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

(Summary So Far)

Relegate any “subjective interpretations” of equality to *named functions*!
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

(Summary So Far)

Relegate any “subjective interpretations” of equality to \textit{named functions} — ideally, in \textit{higher-level} components:
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

(Summary So Far)

Relegate any “subjective interpretations” of equality to *named functions* — ideally, in *higher-level* components:

```cpp
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);
};
```
Relegate any “subjective interpretations” of equality to *named functions* – ideally, in *higher-level* components:

```cpp
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);
};
```
2. Understanding Value Semantics

Value-Semantic Properties

Selecting Salient Attributes

(Summary So Far)

Relegate any “subjective interpretations” of equality to named functions — ideally, in higher-level components:

```cpp
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);
};
```
Relegate any “subjective interpretations” of equality to *named functions* – ideally, in *higher-level* components:

```cpp
class 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);
};
```
2. Understanding Value Semantics

Value-Semantic Properties

A collateral benefit is Terminology:

Saying what we mean facilitates understanding.
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
“...objects are identical...”
“...objects are equal...”
“...objects are equivalent...”
“...create a copy of...”
2. Understanding Value Semantics

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!
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
“...objects are identical...”

(identity)
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

“...objects are identical...”

(.identity)

“...(aliases) refer to the same object...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(value)
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(value)

“...(objects) have the same value...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(value)

“...objects have the same value...”

“...objects refer to the same value...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(value)

“(objects) have the same value...”

“(objects) refer to the same value...”

“(objects) represent the same value...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
“...objects are equal...”
(equality)
2. Understanding Value Semantics

Collateral Benefit: Terminology

“…objects are the same…”
“…objects are equal…”
(equality)
“(objects) compare equal…”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
“...objects are equal...”

(equality)

“...(objects) compare equal...”

“...(homogeneous) operator== returns true...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
“...objects are equal...”

(equality)

“...(objects) compare equal...”

“...(homogeneous) operator== returns true...”

For value-semantic objects:
2. Understanding Value Semantics

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!
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”
(equivalent)
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(equivalent)

In separate \textit{named} functions:
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...” (equivalent)

In separate named functions:

“...fractions are equivalent...”
2. Understanding Value Semantics

Collateral Benefit: Terminology

“...objects are the same...”

(equivalent)

In separate named functions:

“...fractions are equivalent...”

“...graphs are isomorphic...”
2. Understanding Value Semantics

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 \textit{Essential Property of Value}

3. Two Important, Instructional Case Studies
   Specifically, \textit{Regular Expressions} and \textit{Priority Queues}

4. Conclusion
   What \textit{must} be remembered when designing value types
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?
3. Two Important, Instructional Case Studies

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

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

E.g.,

\((1|0)^+\) describes binary numbers.
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*?
Why create a separate class for it?
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.
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;
};
3. Two Important, Instructional Case Studies

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;
};
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;
};

3. Two Important, Instructional Case Studies

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;
};
3. Two Important, Instructional Case Studies

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;
};
3. Two Important, Instructional Case Studies

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;
};
3. Two Important, Instructional Case Studies

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;
};
3. Two Important, Instructional Case Studies

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;
};

What is the value?
3. Two Important, Instructional Case Studies

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;
};
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;
};
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;
};
3. Two Important, Instructional Case Studies

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;
};
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;
};
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!

3. Two Important, Instructional Case Studies

Regular Expressions
3. Two Important, Instructional Case Studies

**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*?
3. Two Important, Instructional Case Studies

Regular Expressions

Does/should it represent a value?
3. Two Important, Instructional Case Studies

Regular Expressions

Does/should it represent a value?

Is a \texttt{RegEx} class a \textit{value type}, or a \textit{mechanism}?
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?
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?
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?
How should its value be defined?

1. The string used to create it.
How should its value be defined?

1. The string used to create it.
2. The language it accepts.
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.
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.
How should its value be defined?

Actually, there is no such accessor, precisely because we defined value the way we did!
How should its value be defined?

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

The correct answer is 2. Why?

Because, there is no accessor to get the string used to assign the value.

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.
How should its value be defined?

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

The correct answer is 2. Why?

Because there is no accessor to get the string used to assign the value.

3. Two Important, Instructional Case Studies

Regular Expressions

Had we provided such an accessor, it would not be considered salient.
How should its value be defined?

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

The correct answer is 2. Why?

Because there is no accessor to get the string used to assign the value.

3. Two Important, Instructional Case Studies

Regular Expressions

Just like capacity for std::vector

Had we provided such an accessor, it would not be considered salient.
3. Two Important, Instructional Case Studies

**Regular Expressions**

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

---

**Perhaps** "Finite Automata" would have been a better name?
3. Two Important, Instructional Case Studies

Regular Expressions

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

```
perhaps

"F I N I T E  A U T O M A T A"
```

"Language"??
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***?
Should such a class be regular?
I.e., Should our `RegEx` class support all of the value-semantic syntax of a *regular* class?
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?
3. Two Important, Instructional Case Studies

Regular Expressions

Should such a class be regular?

Question: How expensive would `operator==` be to implement?
Should such a class be regular?
Question: How expensive would `operator==` be to implement?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?
Should such a class be regular?

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

- \(O[1]?\)
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

- O[1]?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

- \(O[\log N]\)
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 \texttt{operator==} be to implement?

- $O[\log N]$
3. Two Important, Instructional Case Studies

Regular Expressions

Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

- \(O[\log N]\)
- \(O[\sqrt{N}]\)
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?

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \times \log N]$
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \log N]$
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \times \log N]$
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]
- O[N * sqrt N]
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \times \log N]$
- $O[N \times \sqrt{N}]$
- $O[N^2]$
Should such a class be regular?

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

- \( O[\log N] \)
- \( O[\sqrt{N}] \)
- \( O[N] \)
- \( O[N \times \log N] \)
- \( O[N \times \sqrt{N}] \)
- \( O[N^2] \)
- \( O[N^2 \times \log N] \)
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \cdot \log N]$
- $O[N \cdot \sqrt{N}]$
- $O[N^2]$
- $O[N^2 \cdot \log N]$
- Polynomial
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
- $NP$
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \times \log N]$
- $O[N \times \sqrt{N}]$
- $O[N^2]$
- $O[N^2 \times \log N]$
- Polynomial
- NP
- NP complete
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \cdot \log N]$
- $O[N \cdot \sqrt{N}]$
- $O[N^2]$
- $O[N^2 \cdot \log N]$
- Polynomial
- NP
- NP Complete
- P-SPACE
Should such a class be regular?

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

- \(O[\log N]\)
- \(O[\sqrt{N}]\)
- \(O[N]\)
- \(O[N \cdot \log N]\)
- \(O[N \cdot \sqrt{N}]\)
- \(O[N^2]\)
- \(O[N^2 \cdot \log N]\)
- Polynomial
- \(NP\)
- \(NP\) Complete
- \(P\text{-SPACE}\)
- \(P\text{-SPACE Complete}\)
3. Two Important, Instructional Case Studies

Regular Expressions

Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \cdot \log N]$
- $O[N \cdot \sqrt{N}]$
- $O[N^2]$
- $O[N^2 \cdot \log N]$
- 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?

- \(O[\log N]\)
- \(O[\sqrt{N}]\)
- \(O[N]\)
- \(O[N \times \log N]\)
- \(O[N \times \sqrt{N}]\)
- \(O[N^2]\)
- \(O[N^2 \times \log N]\)
- Polynomial
- NP
- NP Complete
- P-SPACE
- **P-SPACE Complete**
- Undecidable

If you just sat down, you were right!
Over an alphabet $\Sigma$, 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 $O[|S| \cdot |T| \cdot |\Sigma|]$. 
Should such a class be regular?

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

- $O[\log N]$
- $O[\sqrt{N}]$
- $O[N]$
- $O[N \cdot \log N]$
- $O[N \cdot \sqrt{N}]$
- $O[N^2]$
- $O[N^2 \cdot \log N]$

Clearly No Equality-Comparison Operators!

Should we avoid value types where equality comparison is expensive?
Should such a class be regular?

Question: How expensive would \texttt{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-
- P-SPACE
- P-SPACE Complete
- Undecidable
3. Two Important, Instructional Case Studies

Regular Expressions

Discussion?
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*?
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*?
What is a *Priority Queue*?
3. Two Important, Instructional Case Studies

Priority Queues

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 *pushes* and *pops* of queue-element values.
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 *pushes* and *pops* of queue-element values.
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

**Example Queue Element:**

```cpp
class LabeledPoint {
    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)
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

Example Queue Element:

```cpp
class LabeledPoint {
    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; }
};
```

Example Comparison Function:

```cpp
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:

```cpp
class LabeledPoint {
    std::string d_label;
    int d_x;
    int d_y;

    // ... (Regular Type)

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

Example Comparison Function:

```cpp
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")
```

4. Two Important, Instructional Case Studies

Priority Queues
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.

Array-Based Heap:

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

Different Priorities, Different Values

Array-Based Heap:
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.

Different Priorities, 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, Different Values

Array-Based Heap:
3. Two Important, Instructional Case Studies

Priority Queues

What is a **Priority Queue**?

Array-Based Heap:

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

Same Priority, Same Value
3. Two Important, Instructional Case Studies

Priority Queues

What is a **Priority Queue**?

Array-Based Heap:

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

**Same Priority, Different Values**
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

Array-Based Heap:

```plaintext
2 3 30 4 80 31 31 99 99 80
```

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

```plaintext
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.

Array-Based Heap:

```
2 3 30 4 80 31 31 99 99 80 2 ...
```
3. Two Important, Instructional Case Studies

Priority Queues

What is a Priority Queue?

Array-Based Heap:

```
2 3 30 4 80 31 31 99 99 80 2 ...
```

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*?

Array-Based Heap:

```
2 3 30 4 2 31 31 99 99 80 80 ...
```

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

```c
q.push(2);
```
3. Two Important, Instructional Case Studies

Priority Queues

What is a **Priority Queue**?

Array-Based Heap:

```
<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>30</th>
<th>4</th>
<th>2</th>
<th>31</th>
<th>31</th>
<th>99</th>
<th>99</th>
<th>80</th>
<th>80</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
```

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*?

Array-Based Heap:

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.

Array-Based Heap:

```
(2 2 30 4 3 31 31 99 99 80 80 ...)
```
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.

Array-Based Heap:

```
2  2  30  4  3  31  31  99  99  80  80
```

```q.push(2);```
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

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

Array-Based Heap:

```
2 2 30 4 3 31 31 99 99 80 80 ...
```

- Each element is labeled with its calculated priority.
- Each distinct color represents an element having a distinct value.
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.

Array-Based Heap:
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

Array-Based Heap:

```
    2  30  4  3  31  31  99  99  80  80 ...
```

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

```q.pop();```
3. Two Important, Instructional Case Studies

Priority Queues

What is a *Priority Queue*?

Array-Based Heap:

- 80
- 2
- 30
- 4
- 3
- 31
- 31
- 99
- 99
- 80

Each distinct color represents an element having a distinct value.

Each element is labeled with its calculated priority.

If equally urgent, choose the left child. Swap with the element having the more urgent priority.

```java
q.pop();
```
What is a *Priority Queue*?

Each element is labeled with its calculated priority.

Each distinct color represents an element having a distinct value.

Array-Based Heap:

```
80  2  30  4  3  31  31  99  99  80
```

$q.pop();$
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.

```latex
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();`
3. Two Important, Instructional Case Studies

Priority Queues

What is a **Priority Queue**?

Array-Based Heap:

```
2 3 30 4 80 31 31 99 99 80 ...
```

Each distinct color represents an element having a distinct value.

Each element is labeled with its calculated priority.

$q.pop();$
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.

Array-Based Heap:

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

```plaintext
2 3 30 4 80 31 31 99 99 80 ...
```
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.

Array-Based Heap:

```
2   3   30   4   80   31   31   99   99   80
```

```q.pop();```
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*?
Why create a separate class for it?
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.
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*?
3. Two Important, Instructional Case Studies

Priority Queues

Does/should it represent a value?
Does/should it represent a value?
Is a PriorityQueue class a value type, or a mechanism?
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?
3. Two Important, Instructional Case Studies

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?
3. Two Important, Instructional Case Studies

Priority Queues

Does/should it represent a value?

I claim, “yes!”

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

Assuming, of course, that the queue-element type is also value semantic.
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*?
How should its value be defined?
3. Two Important, Instructional Case Studies

Priority Queues

How should its value be defined?
How should its value be defined?
How should its value be defined?

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

1. `top`
2. `push`
3. `pop`
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***?
Should such a class be regular?

I.e., should our PriorityQueue class support all of the value-semantic syntax of a regular class?
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?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?
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??
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?

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

Necessary:
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

Necessary:
- Same number of elements.
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

Necessary:
- Same number of elements.
- Same numbers of respective element values.
3. Two Important, Instructional Case Studies

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:
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.
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??
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:

**Array-Based Heap 1:**
```
2 3 30 4 80 31 31 99 99 80 ...
```

**Array-Based Heap 2:**
```
2 3 30 4 80 31 31 99 99 80 ...
```
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

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

Array-Based Heap 1:

\begin{verbatim}
2 3 30 4 80 31 31 99 99 80 ...
\end{verbatim}

Array-Based Heap 2:

\begin{verbatim}
2 3 30 4 80 31 31 99 99 80 ...
\end{verbatim}
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:
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?
Question: How \textit{expensive} would \texttt{operator==} be to implement?

As it turns out, we can distinguish these two values with appropriate \texttt{pushes}, \texttt{tops}, and \texttt{pops}. 
Should such a class be regular?
Question: How expensive would \texttt{operator==} be to implement?

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

But can we always do that?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

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

But can we always do that?

If we aren’t sure, should we implement \texttt{operator==} for this class anyway?
Should such a class be regular?

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

3. Two Important, Instructional Case Studies

Priority Queues

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 not have the same linear heap orderings?

But can we always do that?

If we aren’t sure, should we implement operator== for this class anyway?
3. Two Important, Instructional Case Studies

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?

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 not have the same linear heap orderings?
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 sequence of salient operations that distinguishes them:

What is the complexity of operator==?
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==`?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

Until quite recently, that linear order \textit{is necessary} was just a conjecture.
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.
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

Until quite recently, that linear order \textit{is necessary} was just a conjecture.

I finally have a simple constructive proof.

Here is a \textbf{very} quick sketch:
Should such a class be regular?
Question: How expensive would `operator==` be to implement?
Should such a class be regular?

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

Highest-Index Element Having Distinct Priorities
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

3. Two Important, Instructional Case Studies

Priority Queues

Push Arbitrary Priority-Two Values
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?

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

Push a Priority-One Value
Should such a class be regular?
Question: How \textit{expensive} would \texttt{operator==} be to implement?

### Priority Queues

**Push a Priority-One Value**
Should such a class be regular?
Question: How *expensive* would \texttt{operator==} be to implement?

Priority Queues

Push Arbitrary Priority-Two values
Should such a class be regular?

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

Push a Different Priority-One Value
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

3. Two Important, Instructional Case Studies

Priority Queues

Push a \textbf{Different} Priority-One Value
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?
Question: How expensive would \texttt{operator==} be to implement?

Push $N$ Arbitrary Priority-Two Values
Should such a class be regular?

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

### Priority Queues

#### Push $N$ Arbitrary Priority-Two Values
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

3. Two Important, Instructional Case Studies

Priority Queues

Will Pop First!

Priority One

Priority Two

Will Pop First!

Pop N Elements
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?
Question: How expensive would \texttt{operator==} be to implement?

After almost $N \texttt{pop}$ operations
the tops are not the same!
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

After one more \texttt{pop} operation

the element values are not the same!
Should such a class be regular?

Question: How expensive would $\text{operator}== \text{be to implement}$ be?
Should such a class be regular?

Question: How expensive would \texttt{operator==} be to implement?

\textbf{YES IT SHOULD!}
3. Two Important, Instructional Case Studies

Priority Queues

Should such a class be regular?

Question: How expensive would `operator==` be to implement?
3. Two Important, Instructional Case Studies

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
4. Conclusion

What to Remember about VSTs
4. Conclusion

What to Remember about VSTs

So what are the take-aways?
4. Conclusion

What to Remember about VSTs

So what are the take-aways?

- Some types naturally represent a value.
4. Conclusion

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.
4. Conclusion

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*:
4. Conclusion

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.
4. Conclusion

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*. 
4. Conclusion

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.
4. Conclusion

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
4. Conclusion

What to Remember about VSTs

The **key** take-away:
4. Conclusion

What to Remember about VSTs

The **key** take-away:

What makes a value-type *proper* has essentially **nothing** to do with *syntax*...
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*:
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...
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; **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 j lakos@bloomberg.net

The End