Doc No: N2479=07-0349
Date: 2007-12-09
Author: John Lakos
Bloomberg, L.P.

jlakos@bloomberg.net

Normative Language to Describe Value Copy Semantics

Contents

Motivation

On the fifth day of meetings of the Library Working Group (LWG) in Kona (Friday, October 5, 2007), a defect (issue #684) involving regular expressions (§28) was discussed. This defect was in how the wording of the Working Paper (WP) communicated what was meant by stating that two `std::match_result` objects (§28.10) were "the same." Was it the intention of the author to communicate that the two objects were identical (i.e., at the same address in the same process at the same time)? Or did the author mean that the objects could be distinct as long as they each had certain properties in common? After some discussion, it turned out to be the latter.

So finally the meaning was clear. Or was it? What were those certain properties? The discussion continued. What exactly had to be the same, and how could it be described? A few suggestions were made that were either circular (e.g., "whatever the copy constructor preserves") or incorporated vague or equivocal terminology such as *equals* or *equivalent*. I think I even heard a "definition" that included a phrase like "you know what I mean." Since the defect was "purely in the wording," it was resolved that this issue would be given over to the editor to "fix" (without further review by the full committee).

What it means for two objects to be "the same" is not just an editorial issue; it is an important, pervasive, and recurring concept in practical software engineering: one that fairly deserves the attention of the entire LWG and, ultimately, the C++ user community in general. My solution, based on *value semantics*, is to provide consistent normative language to describe concisely what it means for two objects to be "the same"; hence, this proposal.

Overview

The kernel of my proposed solution – which I will make exact in the ***Proposed Wording*** section – is that there should be a formal entity called the set of *salient attributes* of every type that represents an abstract value and that will implement a copy constructor, etc. Moreover, I propose that every type must document an exact specification of its salient attributes, or else document that it has none.

Salient attributes of a type `T` are a documented sequence of named attributes whose respective values for a given instance of `T` (1) derive from the physical state of only that instance of `T`, and (2) are (typically) accessible via the public interface of `T`. (Often, but not necessarily, the salient attributes correspond to some subset of the data members of `T`.) The definition may be recursive (to a finite extent) in that a documented salient attribute of `T` may itself be of a type `U` having its own salient attributes.

I am proposing that the standard include a common definition of value-semantic types that can be referenced throughout the standard library text in much the same way that the terms "equivalence relation" and "formatted input operation" are used to refer to specific concepts whose definitions need not be repeated at each point of use.

I also list the specific types already in the standard that should be designated as value-semantic types. Note that the purpose of this proposal is not to dictate intended behavior, but merely to communicate it more effectively. If an existing type is "almost" value semantic (i.e., its copy semantics differ slightly from the proposed definition), that type would be characterized as value-semantic with those differences stated explicitly (rather than describing all of its semantics from scratch). In particular, there is nothing in the wording of this proposal that would preclude an allocator from being considered a salient attribute, or that would preclude an allocator that is *not* a salient attribute from being propagated or swapped along with the value.[1]

Finally, I propose changes in the phrasing of the definitions of value-semantic operations to eliminate ambiguous and equivocal phrases such as "objects are equal" and replace them with crisp language that is consistent with the definition of value semantics. Specifically:

- If we want to indicate identity, we say "are the same object" or "refer to the same object" rather than "objects are the same" or "objects are identical".
- If we want to indicate equivalence (abstract equality), we say "objects have the same value" or "objects represent the same value".
- If we want to emphasize the programmatic aspect of a type that has an associated `operator==`, we say "objects compare equal", but never "objects are equal".
- We describe the post-conditions of copy-construction in terms of ***value*** (e.g., "Create an object having the same value as…"), rather than using the nebulous phrase "Create a copy of…"
- We deliberately avoid equivocal phrases such as "objects are equal", "objects are the same", or "objects are identical". Instead, we propose always incorporating the sub-phrase "the same" appropriately (in order to emphasize exactly *what* is *identical*): Two ***aliases*** "refer to (or *are*) ***the same object***", whereas two ***objects*** "refer to (or *have*) ***the same value***."
- In the context of value semantics, we use the phrase "is a value-semantic type" to concisely convey a whole range of related properties, making the defined behavior for these kinds of types easier for readers to understand.

For a more in-depth discussion and justification of why this "value-based" approach is optimal, see the ***Background*** section. For a more detailed presentation of value-semantic properties and value-based classification, see the ***Value Semantics*** section. For my own experience on how best to treat non-salient attributes during, say, copy assignment, see the ***Propagating Non-Salient Attributes*** section.

[1]That is the subject of another proposal.

Conclusion

With the definition of *salient attributes*, the requirement that every *value-semantic* type document its salient attributes, the prescription of exactly how the salient attributes are to be used to define the behavior of copy construction, and the axiomatic properties of value-semantic types, etc., we have both solved the practical problem that we have set out to solve, and also have contributed a precise, technical specification of *value*. I emphasize that this latter accomplishment – the definition of *value* – is of far-reaching practical importance in many aspects of software engineering beyond the main subject of this proposal.

Document Conventions

All section names and numbers are relative to the October 2007 working draft, N2461.

> Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~green strikeouts for deleted text~~ and green underlining for inserted text. Large proposed insertions into the working paper are shown in dark blue.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with yellow shading.

Proposed Wording Changes

To section [definitions] (17.1 Library Definitions), add the following:

> **17.1.??**
> **salient attributes**
> the set of attributes (typically defined operationally by an associated, homogeneous equality operator) whose respective values together comprise the overall (abstract) *value* for an object (irrespective of the details of object state). [Note: *Salient attributes may be represented directly as instance data, or indirectly as derived data, requiring a calculation prior to comparison.* – end note] See [value.properties] 17.??.
>
> **17.1.??**
> **value**
> a notion of a unique abstract entity in a mathematical type system. Two objects of a given C++ type represent (or "have") the same *value* iff each *salient attribute* of one object (recursively) has the same value as the corresponding salient attribute of the other object, where the salient attributes are defined explicitly for each type. See [value.properties] 17.??.
>
> **17.1.??**
> **value-semantic type**
> a C++ type having a publicly accessible default constructor, copy constructor, copy assignment operator, and (typically) equality comparison operator, having a notion of *value* that is defined as the explicitly-specified set of *salient attributes* that must respectively have the same value in order for

two instances of the type to have the same overall value, and adheres to all *value-semantic properties* ([value.properties] 17.??). The associated equality-comparison operator (if it exists), provides an operational definition of value. [Note: *Each of the fundamental and enumerated types has full value-semantics, and pointer types have value semantics within the context of a single running program.* – end note]

Before section [description] (17.3), add the following:

17.?? Value-semantic Properties **[value.properties]**

Within this standard, if a type is described as being a *value-semantic type*, then it adheres to the comprehensive set of axiomatic properties described below. Many of the types in the standard library are value-semantic types. In addition, many class templates in the standard library yield value-semantic types when they are instantiated with value-semantic argument types for their non-optional, template parameters. Each of the fundamental and enumerated types has full value-semantics. Pointer types have value semantics only within the context of a single running program.

Each value-semantic property listed below describes characteristics that must hold for a value-semantic type T, objects *a* and *b* of type T, and an rvalue *rv* of type T.

— An object of type T has an abstract *value*, which is defined by a set of *salient attributes*. Two objects *a* and *b* have the same value iff each salient attribute of *a* has the same value as the corresponding salient attribute of *b*. Type T may also have non-salient attributes, which do not contribute to its value (e.g., `capacity()` for `std::vector<T, A>`).

— Type T has a default constructor, copy constructor, copy assignment operator, `swap(T&,T&)`, and (unless deliberately omitted) an equality-comparison operator that are consistent with all value-semantic properties. [Note: *Under rare circumstances, the associated equality-comparison operator may be deliberately omitted, due either to difficulty in implementation (e.g.,* `std::function`*) or to avoid surprises in program performance (e.g.,* `std::unordered_map`*).* – end note]

— The value of *a* can be changed only through the public interface of T. Modifying an external object that is not owned exclusively by *a* does not change the value of *a*. [Note: *One way to change the value of a through the interface of T is to modify a sub-part of T through a pointer or reference exposed in the interface. The referenced sub-part is not considered an external object.* – end note] Changing the value of an (autonomous) object *b* will not change the value of *a*.

— If two distinct objects *a* and *b* have the same value, and if an identical sequence of operations are applied to both, then (in the absence of exceptions and undefined behavior) the two objects will again have the same value following the application of that sequence to *a* and *b*.

— The expression *a* == *b* returns true iff each salient attribute of *a* has the same value as the corresponding salient attribute of *b* (the values of *a* and *b* are unchanged). Thus, the equality comparison operator provides an *operational* definition of *value* for type T.

— The equality comparison operator defines an equivalence relation, having the qualities of reflexivity, symmetry, and transitivity.

— Given a value-semantic type U (which may be the same as T) and an object *d* of type U, if *a* == *d* is well formed, then *d* == *a* is well formed and returns the same result. Additionally, if *a* == *d* is well formed, then *a* `!=` *d* and *d* `!=` *a* are both well formed and return the same result as `!(`*a* == *d*`)`. [Note: *It is not possible to add a class and cause a pre-existing value-semantic class to cease to be value semantic. For example, suppose we have a value-semantic type* `Point`*, and we subsequently derive a type (e.g.,* `Pixel`*) from* `Point`*.* `Point` *remains a value-semantic type even if* `Pixel` *fails to ensure this property because* `Pixel` *was never a value-semantic type.* – end note]

— The post-condition of the statement, `T` *a*`(`*b*`)` (copy-construction) is that *a* has the same value as *b*. The value of *b* is unchanged.

— The post-condition of the expression, *a* = *b* (copy-assignment) is that *a* has the same value as *b*. The value of *b* is unchanged.

— If a move constructor is supplied for T, the post-condition of the statement `T` *a*`(`*rv*`)` is that *a* has the value initially held by *rv*. (The value of *rv* may change, but *rv* remains destructible.)

— If a move assignment operator is supplied for T, the post-condition of the statement *a* = *rv* is that *a* has the value initially held by *rv*. (The value of *rv* may change, but *rv* remains destructible.)

— If the expression *swap(a, b)* is well-formed, the post-condition is that *a* has the value previously held by *b,* and *b* has the value previously held by *a*. [Note: *The default implementation of* `swap` *in the standard library meets these criteria for any value-semantic type that implements the other operations defined in this section.* – end note]

Specific examples of wording changes for specific library-defined types are as follows:

Make the following changes to section [exception] (18.7.1), paragraph 1:

> The class exception defines the base class for the types of objects thrown as exceptions by C++ Standard library components, and certain expressions, to report errors detected during program execution. The class `exception` and its derived classes are not value-semantic types ([value.properties] 17.??) in that they have no notion of value and no equality comparison. The effects of copy-construction and assignment are implementation-defined.

When a class has a copy constructor and assignment operator, it is easy to infer that it is a value-semantic type. For this reason, it is useful to assert the negative, as in the case of the `exception` class.

Make the following changes to section [pairs] (20.2.3), paragraph 1:

> The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if they were tuple objects (see 20.3.1.3 and 20.3.1.4). An instantiation of `pair<T,U>` is a value-semantic type ([value.properties] 17.??) if `T` and `U` are both value-semantic types.

Here we have an example of a class template that has value semantics only when it is instantiated with value-semantic types.

Insert a paragraph after [container.requirements] (23.1), paragraph 4:

> Objects stored in these components shall be MoveConstructible and MoveAssignable. If the copy constructor of a container is used, objects stored in that container shall be CopyConstructible. If the copy assignment operator of a sequence container is used, objects stored in that container shall be CopyConstructible and CopyAssignable. If the copy assignment operator of an associative container is used, objects stored in that container shall be CopyConstructible.
>
> These containers are value-semantic types ([value.properties] 17.??) when instantiated with value-semantic types for the non-defaulted parameters. The salient attributes of a container that define its value are its `size()` and the values of its elements. Except in the case of unordered containers ([container.unordered]), the order of elements in a container is a salient attribute. [Note: *To avoid performance surprises, unordered containers, which are in all other respects value-semantic types, deliberately do not implement the equality comparison operator.* – end note]

By referring to the central definition of "value-semantic type" and by spelling out the salient attributes, the reader can readily grasp the many essential qualities of copy-construction, assignment, and equality. We also note any deviation from the canonical definition[3] of a value-semantic type (e.g., suppressing the equality comparison operator).

If this proposal is received favorably by the LWG, I will review the entire WP and propose all appropriate changes to existing normative wording in order to effectively communicate the defined behavior of each *value-semantic* standard library component.

Background

As we know, ***copy construction*** is universally interpreted to mean that the resulting object is ***substitutable*** for the original one ***with respect to some criteria.*** In the case where we deliberately have copy construction without an `operator==`, what that criteria might be is anyone's guess, and must be dealt with on a case-by-case basis. On the other hand, if an associated homogeneous (non-member) `operator==` is defined for a type having a copy constructor, there is no ambiguity: `operator==` implements

---

2

3 If we choose to make the propagation (e.g., during copy assignment) of a non-salient attribute part of the defined behavior of a type, we would document that *additional* behavior explicitly. (See the ***Propagating Non-Salient Attributes*** section for when such propagation, in my experience, might be appropriate.)

the criteria for being ***the same***. But what exactly is ***it*** that is "the same?" In other words, what is the contract (i.e., formal specification of defined behavior) for `operator==`?

Same Object?

By "the same," we don't mean that the objects being compared are themselves the same object, for that would imply identity (i.e., they are of the same type and reside simultaneously at the same address in the same program).

Same State?

We also don't necessarily mean that the objects have the same state. For example, two `std::string` objects that compare equal will not necessarily hold the address of the same dynamically allocated memory (even if copy constructed). Two `std::vector<int>` objects may compare equal, even though they can be observed to have different capacities. We can foresee many other situations in which duplicating all of the state would not be appropriate (e.g., the free node pool of a list or tree, the historical data for an adaptive map whose capacity grows based on usage statistics, the address of a mutex for an optionally thread-safe container). Moreover, it is absolutely essential for ***modularity*** that "the same" be described ***abstractly*** in terms of the ***publicly accessible interface***, and not the private data members themselves. Hence, ***two objects that compare equal need not have the same state.***

Same Behavior?

What about behavior? Should it always be true that if one object is copy constructed from another, the two will behave identically from then on? Consider again an `std::vector<int>` containing one element, but having a capacity of 65,536. Is it permissible for the copy-constructed object to have a reduced capacity? If so, the original and copy again have different behavior. If not, then the requirements for a copy-constructed object are different (stronger) than the criteria for being "the same" as defined by `operator==` (which invites the original question of what general substitutability criteria should apply).

Even if we wanted to, requiring the behavior of a copy-constructed object to be substitutable for the original is not possible in general: Common objects such as an `std::string` or an `std::vector` necessarily expose unique state – i.e., the address of the block of memory they manage to represent the data. If the two objects expose different state, we can easily write code that exploits these differences in such a way that the two objects would not yield the same behavior. (Note that, by definition, distinct objects that exist simultaneously within a given program reside at different addresses in memory, which alone is sufficient to contrive software that will distinguish the two objects.)

As a practical matter, propagating certain kinds of behaviors during copy construction can be highly undesirable. For example, given a container object that can optionally be made thread-safe by passing it the address of a particular mutex at construction, should that associated mutex necessarily be propagated when that object is passed (by `const` reference) into a function that makes a local copy? If so, then (1) the local object will incur the same synchronization overhead as the original, even though it is necessarily accessed by only a single thread, and (2) any subsequent attempt to access the two objects simultaneously (e.g., assignment) could result in deadlock!

We could attempt to work around the mutex-propagation problem by avoiding the copy constructor, but there is no general way to avoid copy-construction of temporaries. We can foresee many other situations (e.g., region memory allocators) in which propagating an instance-specific mechanism that is clearly orthogonal to any abstract notion of value would be sub-optimal, dangerous, or just wrong. The take-away message here is that copy-construction need not result in an object that behaves the same as the original in all circumstances, so long as the ***copy satisfies all*** of the ***defined behavior*** advertised in the contract for an object of that type.

Same Value!

If not identical behavior, what then? (The answer had better be easy to understand and apply if human beings are to benefit.) What I propose *must* be "the same" after copy construction is "whatever the associated homogeneous ***equality comparison operator defines*** *(documents)* to be the ***salient attributes*** for that type" – i.e., "the specific attributes that must respectively compare equal in order for the objects as a whole to compare equal." Hence, the (observable) *values* of these *salient attributes*, and not the raw instance state used to represent them, comprise what we call the ***value*** of the object.

Let's take a step back and see if this notion of *value* matches our intuition. For example, consider a simple `Date` class with its salient (integer) attributes identified as *year*, *month*, and *day*. Two `Date` objects ***have the same value*** (and therefore compare equal) if the ***respective salient attribute values*** returned by `year()`, `month()`, and `day()` are ***the same***, irrespective of any differences in internal representation (i.e., state) between these two objects:

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

Returns: `true` if the specified `lhs` and `rhs` date objects have the same value, and `false` otherwise. Two dates have the same value if and only if the respective values for `year()`, `month()`, and `day()` are the same.

As a second example, consider an `std::vector<T, A>`, which has as its *salient attributes* its *size* and the *sequence of T values* it represents (but not other *observable attributes* such as its *capacity*, its memory-allocator *instance* (of type `A`), or the *address* of the block of dynamic memory it manages):

```
template <class T, class A>
bool operator==(const vector<T, A>& lhs,
                const vector<T, A>& rhs);
```

Returns: true if the specified `lhs` and `rhs` vector objects have the same value, and false otherwise. Two vectors have the same value if they have the same `size()`, and each element in one has the same value as the corresponding element in the other.

What I have suggested puts the ***burden of defining*** what publicly observable features of an object's state are relevant to the ***substitutability criteria*** governing *copy construction* ***onto*** the ***equality comparison operator*** rather than onto the *copy constructor* itself, thereby leaving room for a copy constructor ***not*** to replicate all observable features and behaviors. A corollary to this proposal is that every class that has a copy constructor should also have an associated homogeneous `operator==` (or, at a minimum, find some way to specify the *salient attributes* for that type). For well-designed classes, what is and is not a *salient attribute* will (or should) be unsurprising and, ideally, obvious and intuitive. In any event, the homogeneous ***`operator==`*** for a given type provides the unequivocal ***operational definition*** of *value*, defining exactly ***what*** must be *the same* in order for the copy-constructed object to be considered *substitutable* for the original.

Value Semantics

The meaning of *value* that we want to establish has some grounding in mathematics. A ***mathematical type*** consists of a set of ***globally unique value*****s** – each one describable independently of any representation. (E.g., 512 and 2^9 are two representations of the same value.) A C++ type that properly represents (a subset of the values of) a mathematical type is said to have value semantics. By virtue of this (partial) mapping from C++ types to abstract (i.e., representation-independent) mathematical ones, we see that a C++ object that "has" a value is just another instance of a representation that refers to the corresponding *globally-unique value*. This ***platonic*** notion of *value* helps us avoid equivocal phrases like "objects are the same", "objects are equal", or "objects are equivalent" by identifying the ***substitutability criteria*** explicitly: "objects have the same value." It follows that to copy a value-semantic object is to cause another object to be created that ***has the same value*** as the original.

All of the fundamental, enumeration, and pointer types in C++ have value semantics, as do most types defined in the standard library. However, not every type of object that has *state* has an obvious notion of *value* that can be articulated independently of its representation within a running program. Objects that implement useful ***mechanisms*** such as scoped guards, TCP/IP sockets, or thread pools, all have *state*, but have no abstract (i.e., representation-independent) notion of *value,* and are therefore not value-semantic types. If we are unable to articulate the copy-constructor's contract, we won't be able to test it either. Hence, another corollary is that if a type has ***no*** notion of ***value***, it should ***not*** have a ***copy-assignment operator***, an associated ***equality operator***, or, with few exceptions[4], a ***copy constructor***.

Value-Semantic Properties

There is a comprehensive, consistent, and (mostly) intuitive set of properties that apply to C++ types that properly represent *values*. Each of these properties is objectively verifiable, independent of the application domain.[5] 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 we expect the following properties to hold:

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

- `operator==(T, T)` describes an equivalence relationship:
    - $a == a$ (reflexive)
    - $a == b \Leftrightarrow b == a$ (symmetric)

[4]E.g., **exception classes** (which are handled specially by the compiler as they are propagated up the program stack), **function objects** (which do have a notion of value, but, due to *type erasure,* equality comparison is difficult to implement), and **output iterators** (for which the copy is not independent of the original).

[5] Many of these properties are also described by Alexander Stepanov as being the properties of a *regular type*. A value-semantic type is different from a regular type in that the former is required to have some notion of an ***abstract value***.

- $a == b$ && $b == c \rightarrow a == c$ (transitive)
- $!(a == b) \Leftrightarrow a\ != b$
- $a == d$ (compiles) $\Leftrightarrow$ $d == a$ (compiles) Note that d is of some *other* type.

- The *value* of *a* is independent of any external object or state; any change to *a* must be accomplished via *a*'s (public) interface.
- Having a copy constructor implies the notion of *value*, and strongly suggests the existence of an associated `operator==`.
- After copy construction, both objects have the same value.
  – i.e., `T b(a);   assert(b == a);`
- If $a == b$ is true initially, then $a == b$ must still be true after applying the same arbitrary sequence of operations to each object (in the absence of exceptions[6] or undefined behavior).

  – e.g.,
```
int x = a;       int y = a;     assert(x == y);
x *= 50;         y *= 50;       assert(x == y);
x %= -2 ;        y %= -2;       assert(x == y);
```

  Note this property does not necessarily hold for value-semantic objects of different types:

  – e.g.,
```
short x = short(a);    int y = short(a);    assert(x == y);
x *= 50;               y *= 50;             assert(x ?? y);
```

- a == b $\Leftrightarrow$ The *salient attributes* (i.e., all attributes that contribute to overall value) respectively compare equal.

  – e.g.,
```
class Point { int x, y; public: Point();
                                void setX(int x);  int x() const
                                void setY(int y);  int y() const; };
Point a, b;
// …
if (a == b) assert(a.x() == b.x() && a.y() == b.y());
```

[6] In particular, the same memory allocation mechanisms can be shared by two objects of a given type; this mechanism may succeed for an operation on one object and then subsequently fail for that same operation on the other (even though they may have initially had the same value).

- Having an associated `operator==` implies the meaning (and strongly suggests the existence) of both a copy constructor and a copy assignment operator.

- After copy assignment, both objects have the same value.
  – i.e., `a = b;   assert(a == b);`

- Given some other constructor (e.g., the default constructor) and the copy assignment operator, we can synthesize the logical behavior of copy construction by first creating the object with an arbitrary value and then assigning it the desired value.
  – i.e., `T a; a = b;   T c(b);   assert(c == a);`

A Value-Based Classification Hierarchy

I have found it useful (both for human cognition and effective testing) to partition types that have instance state into a small number of classifications, based on whether or not they support the notion of a *value*:

- **Full value-semantic types:** These are types, such as `int`, `std::string`, and `std::vector<int>`, that represent *globally unique values* that can be articulated and understood *outside* of the context of a *running program*.

- **In-core value-semantic types:** These are types such as pointers and iterators, whose value incorporates the identity (e.g., address) of some other autonomous object. For example, two iterators have the same value if they simultaneously refer to the same (identical) object. Because this notion of value incorporates a process-specific memory address, objects having *in-core value semantics* have *no globally unique value*, and hence cannot be (independently) externalized (e.g., `int*`).

- **Reference-semantic types:** These are types such as C++ references and proxy classes that look a lot like value-semantic types (in that they have copy constructors and may have assignment and equality operators), but that do not have value-semantic properties. In particular, the copy constructor for a reference type operates on the reference object itself, but the assignment and equality operators (if they exist) operate on the *referred-to* object (assuming it has a notion of value). Reference-semantic types are also typically immutable[7]: Each

[7] For example, the fact that C++ references cannot be rebound allows programmers and compilers to reason about the code and allows for important compiler optimizations.

instance of a reference type is bound to a specific object at construction, and continues to refer to that same object throughout the lifetime of the reference object.

- **Mechanism types:** These are types such as output streams and thread pools that have *no abstract notion of value*, and therefore should not have copy constructors, equality operators, or copy-assignment operators.

The C++ language provides software developers with a vast design space. Without a sound framework to provide guidance, many otherwise similar types will be created, having subtle and gratuitous differences in their semantic properties. There is an affirmative benefit to deliberately limiting ourselves to just a few well-chosen categories of types. Within a category, each type is characterized by *the same* comprehensive set of well-defined, cohesive properties. These properties can be invoked by simply naming the category, rather than having to repeatedly provide a comprehensive, detailed description of every operation for every type. These specific categories and the regularity they impose make the resulting software easier to describe, understand, develop, test, use, and maintain.

Propagation of Non-Salient Attributes

This final section was added[8] in order to provide guidance regarding whether an observable attribute that does not contribute to the overall value of a value-semantic type should be propagated (e.g., via copy construction) or swapped along with the salient attributes. By default, the propagation characteristics of a non-salient attribute are unspecified. If the propagation characteristic of such an attribute is part of defined behavior, it must be specified explicitly. What follows derives from practical experience, but is otherwise unrelated to this proposal.

For types that are categorized as value-semantic, I propose that there are three classifications of object state:

1. State that contributes to *value* - e.g., the size of an std::vector.

2. State that does not contribute to *value*, yet is not independent of value - e.g., the capacity of an std::vector.

[8] After reviewing an earlier version of this proposal, Bjarne Stroustrup asked me to provide specific examples of when it might or might not be appropriate to copy state that does not contribute to value.

3. State that is *orthogonal* to *value* (i.e., independent of the individual values of all *salient attributes*) - e.g., a lock, an allocator, usage statistics.

Object state in category #1 *must* be copied, assigned, swapped, etc. Object state in category #2 *may* or may *not* be copied. Typically such state is part of the implementation of the mechanism that helps to represent the value and can be treated as an implementation detail. Object state in category #3 should generally *not* be copied, assigned, swapped, etc. This is state that typically has a default, but can be configured via additional arguments supplied by the client at construction. Once configured (e.g., by the address of some external mechanism), the configuration typically cannot change for the life of the object. If this client-specified configuration is intended to be instance-specific, non-propagation would necessarily be part of defined behavior, and so documented.

Again, all of this philosophy applies *only* if the phrase "value-semantic type" is invoked. If not, we're back to square one, and nothing relating to this proposal pertains. But because many types are naturally value-semantic, we can conveniently establish this baseline definition with which everyone will be familiar. We can describe any *differences* in semantics easily and effectively, rather than each time trying describe *all* of the semantic properties from scratch.