Canonical Structs

Daniel B Davidson

13 November 2012

1 Purpose

The purpose of this tech note is to outline some of what a designer of *structs* needs to consider and specifies some guidelines to ease the effort. It also highlights some of the difficulties a `C++` programmer may have when using `D` *structs*.

It is intended to be *opinionated*. There are often many ways to *skin a cat* and the general approach preferred here is to find one way that works well consistently and stick to it. There may be better ways.

Along the way several didactic *structs* are developed, starting simple and building to somewhat complex. It covers equality, comparison, hashing, basic operator overloading and establishes a philosophy for using *structs*. This document only deals with *structs*, *classes* are completely ignored.

2 Deep vs Shallow Semantics

TDPL does a good job describing the differences between *classes* and *structs* for this. Please review TDPL 7.1. The deep versus shallow semantics of a *struct* come into play whenever you do any of the following:

- *Copying*: Instances are copied from one place to another through assignment and copy construction.
- *Comparing*: There are two types of comparison: equality comparison and ordered comparison. In the first case the goal is to find out if the instances are logically the same. The latter is to determine a relative order between two instances, e.g. the first instance is less than the second.
- *Hashing*: Hashing of keys is required to store data in the built-in associative array. The idea of a hash is to examine some or all of the data in an instance to compute a single hash code. This code can then refine a search and greatly reduce the number of comparisons required to lookup a value by key.

In terms of deep versus shallow semantics the characteristics for the default behavior of copying, comparison, and hashing are fairly consistent, but in some cases distinctions between them are necessary. In these cases, the terms *Copy Semantics*, *Compare Semantics* and *Hash Semantics* will differentiate.

A *struct* has *deep semantics* if the operations involved behave as expected, without the surprises introduced by data sharing, most typically through *aliasing*. Otherwise, the *struct* has *shallow semantics*. An example of potentially surprising behavior resulting from data sharing (see TDPL for more examples):

```
import std.stdio;
void main() {
  int[] my_data = [1,2,3];
  int[] copied_data = my_data;
  my_data[0]++;
  writeln(copied_data);
}
```

In this example the output is `[2, 2, 3]`. To the uninitiated, this might be surprising behavior since on the surface it looks like you have successfully copied the data when in reality you have only copied the pointer and are actually sharing data.

Copy semantics behave with the fewest surprises and therefore should be preferred.

With that said, there are times when reference semantics do make sense and have benefits. The typical case is when sharing of data reduces memory and the unnecessary creation of objects. One of the best examples in `D` is the string. The string is really `immutable(char)[]` and is semantically special (see below).

2.1 *Clucts* Hybrid Types *TDPL 1.7 and 7.1.3*

A *cluct* is any *struct* that has any of:

1. Pointer
2. Dynamic array with reference type that is mutable
3. Associative array
4. OR any user defined structure with 1, 2, or 3 recursively

It is important to identify these up front because unexpected sharing of data or potentially invalid object comparison will occur if not handled properly.

When defining a *struct*, as soon as you know you have one that is a *cluct* you need to decide whether you want it to be a:

- *value cluct*: which is a *cluct* that overcomes the sharing issues by doing deep copies of all of its **direct** members that are *reference clucts* and regaining its value semantics.
- *reference cluct*: which does not have the deep copy semantics and permits data sharing.

Again, according to the *principle of least astonishment*, going with *value cluct* is preferred.

2.1.1 Builtin Arrays

The builtin dynamic array is effectively a *reference cluct*. Copying them by assignment (i.e. without *dup*) inherently shares data. The purpose of the *dup* methods is then to overcome this and actually do deep copies of *reference clucts*.

There is one subtle aspect to the builtin dynamic array **if** the type stored in the array is **not mutable**. In this case that dynamic array of **immutable**, in the context of *copy semantics* can be considered a *value cluct*, because while copying one `immutable(T)[]` to another does cause data sharing, the magic of the dynamic array with **immutable** elements is that such sharing of data can never cause changes in one instance that impact the other. So, for example, one nice aspect of `string`, which is actually the type `immutable(char)[]`, when used in a `struct` is you never need to worry about sharing even though it is happening. Therefore, it effectively still has value semantics *in terms of data sharing*.

```
import std.stdio;
import std.string;
struct S { immutable(char)[] s; }
void main() {
  string str1 = "foo".idup;
  string str2 = "foo".idup;
  assert(str1.ptr != str2.ptr && str1 == str2);
  S s1 = {str1};
  S s2 = s1;
  // This shows clearly that the pointer is being copied, not the data
  assert(s1.s.ptr == s2.s.ptr && s1==s2);
  s1.s = "goo";
  // Changing data does not cause issues
  assert(s1.s == "goo" && s2.s == "foo");
```

```
  s1.s = "foo";
  assert(s1.s == "foo" && s2.s == "foo");
  writeln("pointers␣", s1.s.ptr, "␣", s2.s.ptr, "␣equal␣", s1==s2);
}
```

In this example `S.s` is typed as `immutable(char)[]` which is the same as a `string`. Equality comparing arrays of `immutable(T)[]` behaves as you would expect, each element in the array is compared in turn and if any are different the result is false. Include this `immutable(T)[]` into a *struct* and the *copy semantics* are preserved in that while there is sharing (i.e. `s1.s.ptr` equals `s2.s.ptr`) that sharing ends as soon as either modifies the data, which it can do only by modifying the array itself *in-toto* and not the elements.

Note the last line in the example that does the `writeln`. Example output:

`pointers 44E3FD 7FD47378BFF0 equal false`

Therefore, sharing can be happening just fine and as soon as a change would affect both the one getting changed gets a new piece of memory and the other is left in tact.

> Dynamic arrays of immutable type should be preferred to mutable if mutation is rarely needed.

The benefit comes with the safe sharing of data and reduction of unnecessary copying available to `immutable(T)[]` and not to `T[]`.

This shows that the strings now point to separate arrays. Unfortunately, the structs, while having the same logical state are, using default equality comparison, not equal. This may or may not be a bug. For detailed discussion visit: http://d.puremagic.com/issues/show_bug.cgi?id=3789 No matter what the final verdict on that issue, it seems safest to take `opEquals` behavior into your own hands whenever there is *reference semantics* even if it is of the `immutable(T)[]` variety.

The same **immutable** characteristic does not apply to *associative arrays*. Even if both the key type and value type are **immutable**, the default assignment of the associative array still carries reference semantics.

```
import std.stdio;

void main() {
  string[string] aa;
  aa["foo"] = "bar";
  string[string] bb = aa;
  aa["goo"] = "moo";
  writeln(bb, "␣", bb == aa);
}
```

In this case `aa` and `bb` are equal even though it looks like only `aa` was updated after the copy. Therefore, all *associative arrays* are *reference clucts*.

2.1.2 *Value Cluct*

To ensure that a newly designed *cluct* is a *value cluct* carrying *copy value semantics* define a `postblit` (i.e. a *this(this)* method), giving it proper deep copy semantics. In the `postblit` implementation you must *dup* every field that is a *reference cluct*. This includes all associative arrays, all dynamic arrays whose type is **mutable** and any *cluct* that has not gone the extra mile to become a *value cluct* (i.e. any *reference cluct*). There is a nice feature of the way `postblit` works that allows you to focus only on the members that need to be deep copied (i.e. *dupped*). The invocation of the copy instantiation of a struct is by default a simple byte copy (i.e. a `blit`). The `postblit` is just your opportunity to turn any potential shallow copies into deep ones via *dup*.

To ensure that the newly designed *cluct* carries *comparison value semantics* define a suitable `opEquals`. Unfortunately, there is no similar feature with `opEquals` allowing you to focus on only the direct members with *reference semantics*. If you implement `opEquals` you need to include all logical fields, recursively.

To be clear, direct members that are *value clucts* can be glossed over in the `postblit` when it comes to *copy semantics*, but not when it comes to *equality comparison semantics*. According to TDPL "The `postblit` constructor calls are consistently inserted whenever you copy objects, whether or not a named variable is explicitly created". Since all fields which are *value clucts* have implemented a proper `postblit` you can be sure when implementing your own *struct's* `postblit` that those fields which are *value clucts* have been deep copied.

2.1.3 *Reference Cluct*

To allow a *cluct* to remain a *reference cluct* simply don't implement the `postblit`.

If you do this it is a good idea to ensure this *cluct* is never used as a key to hash. The problem if you do is the hashing function might create unequal hash-codes for objects you expect to be deep equal.

Similarly, you would ideally ensure that *equal comparison* is never used. Remember, `is` is the way to tell if two items refer to the same object. *opEquals* is the method intended to determine if two objects are logically or state equivalent. If those objects contain pointers the meaning of the comparison changes significantly since logically equivalent objects can compare unequal.

```
import std.stdio;
import std.string;

void main() {
  struct S { string[string] m; }
  S s1 = {["foo" : "bar"]};
  S s2 = {["foo" : "bar"]};
  writeln(s1 == s2);   // Oooop - looks equal but returns false
}
```

Consider implementing a *dup* method. This is not a requirement, but a convention.

2.1.4 *Value Cluct* and *Reference Cluct* Examples

- Example: Not a *Cluct*

  The following struct is not a *cluct*. There are no pointers in the *struct* or any members it contains (recursively).

```
// A is not a cluct since it only has a static array
struct A {
  int[16] data;
}

// This is also not a cluct since it (and its members recursively) have no
// reference clucts (in fact no clucts at all).
struct NotACluct {
  int i;
  double d;
  A a;
}
```

- Example: Obvious *Reference Cluct*

```
import std.stdio;
struct ObviousReferenceCluct {
  // This is a reference cluct because it has dynamic array with mutable type
  // char and has not implemented a postblit with deep semantics
  char[] _data;
}
```

```
void main() {
  writeln(ObviousReferenceCluct(['a']) == ObviousReferenceCluct(['a'].dup));
  string[ObviousReferenceCluct] aa;
  aa[ObviousReferenceCluct(['a'])] = "foo";
  writeln(aa);
}
```

This *struct* is obviously a *reference cluct* since it has a dynamic array of element type that is **mutable**. Therefore it has reference semantics. This may not be bad, depending on what you want, but there are now certain things you don't want done. For example, you don't want it to be hashed because two instances you would consider equal would have different hash codes. As written, the example allows equality comparison and hashing.

It should not and a way to disable them is shown below.

```
import std.stdio;
struct ObviousReferenceCluct {
  // This is a reference cluct because it has dynamic array with mutable type
  // char and has not implemented a postblit with deep semantics
  char[] _data;
  hash_t toHash() const /* pure */ nothrow {
    static assert(0, "ObviousReferenceCluct can not be used as key to a hash");
  }
  bool opEquals(const ref ObviousReferenceCluct) {
    static assert(0, "ObviousReferenceCluct can not be compared for equality");
  }
  bool opEquals(const(ObviousReferenceCluct)) {
    static assert(0, "ObviousReferenceCluct can not be compared for equality");
  }
}

void main() {
  // Now not allowed
  writeln(ObviousReferenceCluct(['a']) == ObviousReferenceCluct(['a'].dup));
  // Now not allowed
  string[ObviousReferenceCluct] aa;
  aa[ObviousReferenceCluct(['a'])] = "foo";
  writeln(aa);
}
```

- Example: Now a *Value Cluct*

```
struct ValueCluct {
  // This is a value cluct b/c it has dynamic array with mutable type char,
  // but it also has a postblit copying all reference clucts (i.e. _data)
  char[] _data;
  this(this) {
    _data = _data.dup;
  }
}
```

This *struct* has a suitable `postblit` that copies all *reference cluct* members (i.e. its `_data`). In terms of *copy semantics* it has now been "promoted" to a *value cluct*. In terms of *comparison semantics* it still falls short. The shortcoming is addressed as follows:

```
struct ValueCluct {
  // This is a value cluct b/c it has dynamic array with mutable type char,
  // but it also has a postblit copying all reference clucts (i.e. _data)
  char[] _data;
  this(this) {
    _data = _data.dup;
  }
```

```
  bool opEquals(const ref ValueCluct other) const {
    return _data == other._data;
  }
}

void main() {
  ValueCluct vc = {['a'].dup};
  ValueCluct vc2 = vc;
  assert(vc._data.ptr != vc2._data.ptr && vc == vc2);
}
```

The example now includes a suitable `opEquals` restoring *comparison semantics*.

- Example: Not So Obvious *Reference Cluct*

```
struct A { int[] int_a; }
struct B { A a; }
struct C { B b; }
struct D { C c; }
struct ReferenceCluct {
  string s;
  int i;
  D d;
}
```

This *struct* is less obviously a *reference cluct*. Even so, here it is fairly easy to see: `struct A` includes a dynamic array, which has reference semantics. As it stands, without additional methods being implemented, `B`, `C`, `D`, and `ReferenceCluct` all have reference semantics. But imagine that `struct A` were pulled in from some other module. Further, imagine that `struct A` had many other *value cluct* members, lots of comments, many methods to wade through, and deep down this one member with reference semantics, causing a chain of reference semantics all the way up.

2.1.5 Transparency of Semantics

In the last example a user of a `struct` might be forgiven for thinking it had value semantics. The problem is there is no transparent indication that a *struct* has reference or value semantics. But whenever instances are copied, assigned, compared, or hashed it is a critical distinction. By default, copies and assignments of *reference clucts* leads to shared data. Similarly, by default, comparison and hashing are not deep and inherently are dealing with pointers, which leads to surprises (i.e. bugs).

In the *not so obvious reference cluct* example, assume the `int_a` field in `struct A` were newly introduced in the latest code branch about to be mainlined. Further, assume there are many more members at each level, but all with good *value semantics*. At the time this *reference cluct* is introduced in `struct A`, without any supporting code changes the meaning of each of these changes for all `A`, `B`, `C`, `D`, and `ReferenceCluct`:

- Copy Initialization and Assignment: By default copy initialization is a byte copy from the source to the target (i.e. a blit). With the introduction of `int_a` a byte copy no longer suffices to keep *deep copy semantics*. After copying one instance of `ReferenceCluct` to another, changes in `int_a` are seen by both.

- Equality Comparison: Equality comparison of objects with reference semantics is error prone. Before the introduction of `int_a` everything was fine with `opEquals` since, by default it does a byte by byte comparison. With the introduction of `int_a` this comparison is likely not what is desired or expected, since the comparison of `int_a` will be comparing the pointers associated with the array instead of the elements.

- Hashing: Default hashing for *structs* is byte by byte. This is convenient but with the introduction of `int_a` the mechanism no longer has the same meaning. Two `ReferenceCluct` instances that both have `int_a` with contents `[1,2,3]` and all other fields equal will have different hash codes.

This change in semantics and therefore the fundamental behavior by the introduction of a single field to a *struct* is troubling.

- Repairing the Damage

  Both *Copy Semantics* and *Equality Semantics* need to be repaired after the introduction of `int_a`. Repairing *copy semantics* is more straightforward than repairing *equality semantics* due to the **consistent** insertion of `postblit`.

  In this case, repairing the opEquals just in the `struct A` which has changed is enough. This is because `postblits` are called consistently. However, the same is not true for equality comparison. Introducing a suitable `opEquals` into `struct A` is necessary but not sufficient for equality to carry through to composing *struct*.

```
import std.stdio;
struct A {
  int[] int_a;
  this(this) { int_a = int_a.dup; }
  bool opEquals(const ref A other) { return int_a == other.int_a; }
}
struct B { A a; }
struct C { B b; }
struct D { C c; }
struct ReferenceCluct {
  string s;
  int i;
  D d;
}

void main() {
  ReferenceCluct rc1;
  rc1.d.c.b.a.int_a = [1,2,3];
  ReferenceCluct rc2 = rc1;
  assert(rc1.d.c.b.a.int_a.ptr != rc2.d.c.b.a.int_a.ptr);
  assert(rc1.d.c.b.a == rc2.d.c.b.a);
  writeln("Oops␣-␣composing␣struct␣comparison␣=>␣", rc1 == rc2);
}
```

  This example outputs: `Oops - composing struct comparison => false`

  So, for `ReferenceCluct` to be deep equality comparable, each among `A`, `B`, `C`, `D` and `ReferenceCluct` must now have a suitable `opEquals`.

```
import std.stdio;
struct A {
  int[] int_a;
  this(this) { int_a = int_a.dup; }
  bool opEquals(const ref A other) { return (this is other) || int_a == other.int_a; }
}
struct B {
  A a;
  bool opEquals(const ref B other) { return (this is other) || a == other.a; }
}
struct C {
  B b;
  bool opEquals(const ref C other) { return (this is other) || b == other.b; }
}
struct D {
  C c;
  bool opEquals(const ref D other) { return (this is other) || c == other.c; }
}
struct ReferenceCluct {
  string s;
  int i;
```

```
  D d;
  bool opEquals(const ref ReferenceCluct other) {
    return ((this is other) ||
            (s == other.s &&
             i == other.i &&
             d == other.d));
  }
}

void main() {
  ReferenceCluct rc1;
  rc1.d.c.b.a.int_a = [1,2,3];
  ReferenceCluct rc2 = rc1;
  assert(rc1.d.c.b.a.int_a.ptr != rc2.d.c.b.a.int_a.ptr);
  assert(rc1.d.c.b.a == rc2.d.c.b.a);
  writeln("Ahhh, that's better => ", rc1 == rc2);
}
```

Now both deep copy and deep equality comparison semantics are in tact, as this prints:

> Ahhh, that's better => true

- Contrast with C++

This situation is very similar to `C++`. In `C++` you would have to implement an `operator==` for each of `A`, `B`, `C`, `D`, and `ReferenceCluct`. The struct `A` might look something like:

```
struct A {
  typedef std::vector< int > Int_vector_t;
  Int_vector_t int_a_;
  bool operator==(A const& rhs) const {
    return
      ((this==&rhs) or (
        (int_a_ == rhs.int_a_)));
  }
};
struct B {
  A a_;
  bool operator==(B const& rhs) const {
    return ((this==&rhs) or ((a_ == rhs.a_))); }
};
...
```

Note, here the same machinations for equality comparison are required. But, no equivalent *dup* call is needed for the `Int_vector_t` field because the standard collections already do the deep copy. It is not that in `C++` you somehow get away without dealing with copy semantics, whereas in `D` you have to always be thinking of adding in the `postblit`. It is more that in D, for dynamic and associative arrays they decided to go with shallow semantics and one field in the hierarchy of your struct with shallow semantics requires you to deal with it. In `C++` if all you use is fundamental types, strings, and container classes without pointers then the issues of semantics are there, they have just been addressed by your selection of what to use.

- Generalizing

Either way, in both `D` and `C++`, the introduction of new fields with *reference semantics* into a *struct* needs to be dealt with by that *struct* and all composing structs. As the number of fields increases so does the required code. There two excellent features in `D` that, when combined, can greatly help to reduce boilerplate code, and those are `compile time reflection` and the `mixin`.

2.1.6 Mixin To The Rescue

Wouldn't it be nice if all this boilerplate code could be wrapped in a simple function. Take `opEquals`, for instance. Essentially the pseudo-logic is straightforward:

```
bool opEquals(const ref T other) const {
  foreach(field, fieldsOfStruct) {
    if(this.field != other.field) {
      return false;
    }
  }
  return true;
}
```

`D` does provide a mechanism to get at all fields of a *struct* at compile time with `T.tupleof`. Given any type `T`, `T.tupleof` gives a range of types of the fields in the *struct* that can be iterated on. For example:

```
struct S {
  int someInt;
  string someString;
  double someDouble;
}

void main() {
  foreach(i, t; typeof(S.tupleof)) {
    pragma(msg, "Found␣field␣named␣",
           S.tupleof[i].stringof, "␣of␣type␣", t);
  }
}
```

This produces:

```
Found field named (S).someInt of type int
Found field named (S).someString of type string
Found field named (S).someDouble of type double
```

Back to `opEquals`. Clearly there is a recursive nature to the general comparison function. Comparing a type `ReferenceCluct` deeply requires comparing a type `D`, which requires comparing a type `C`, which requires, ... etc. Rather than attempt to do the function as a member, it is simpler to have a global function and make use of basic recursion. A new outline of the logic follows:

```
bool typesDeepEqual(T,F)(auto ref T lhs, auto ref F rhs) {
    bool result = true;
    if(lhs is rhs) { return true; }

    static if(isPointer!(T)) {
      if(lhs && rhs) {
        result &= typesDeepEqual(*lhs, *rhs);
      } else {
        result = !(lhs || rhs);
      }
    } else static if(is(T == struct)) {
      // iterate over fields and
      foreach (i, ignore ; typeof(T.tupleof)) {
        result &= typesDeepEqual(lhs.tupleof[i], rhs.tupleof[i]);
        if(!result) return false;
      }
    } else static if(isAssociativeArray!(T)) {
      //... compare keys and values
    } else static if(isDynamicArray!(T)) {
      // ... compare elements
    } else {
      // Assume existing comparison works
      result = lhs == rhs;
    }
```

```
    return result;
  }
```

We know we need special treatment of embedded pointers and *structs* because the default is byte comparison. To compensate, for pointers we must drill down into the pointed to objects and compare them. Similarly, for structs we must drill down into the fields and compare them. Additional drilling down must be done for dynamic arrays and associative arrays. For the implementation see `opmix.mix.typesDeepEqual`.

Now that there is a global function making use of `compile time reflection`, we need a convenient way to pull that into the *struct*. The mixin offers this capability.

```
const string OpEquals = `
bool opEquals(const ref typeof(this) other) const {
  return typesDeepEqual(this, other);
}
```

In this example the `OpEquals` is a compile time string that can be included, much like a macro in `C++`, into a *struct* to provide a bit of code generation. The updated `ReferenceCluct` example with the `opEquals` generated now looks like the following. All output remains the same, only now there is no need to keep the `opEquals` in sync with the *struct* manually.

```
import std.stdio;
import opmix.mix;
struct A {
  mixin(OpEquals);
  int[] int_a;
  this(this) { int_a = int_a.dup; }
}
struct B {
  mixin(OpEquals);
  A a;
}
struct C {
  mixin(OpEquals);
  B b;
}
struct D {
  mixin(OpEquals);
  C c;
}
struct ReferenceCluct {
  mixin(OpEquals);
  string s;
  int i;
  D d;
}

void main() {
  ReferenceCluct rc1;
  rc1.d.c.b.a.int_a = [1,2,3];
  ReferenceCluct rc2 = rc1;
  assert(rc1.d.c.b.a.int_a.ptr != rc2.d.c.b.a.int_a.ptr);
  assert(rc1.d.c.b.a == rc2.d.c.b.a);
  writeln("Ahhh, that's better => ", rc1 == rc2);
}
```

The same approach can be used to define a `postblit` that *dups* all fields that require it. The nice thing about the `postblit` is that since they are called consistently, we only need to worry about the fields in the current struct, not the fields of any structs recursively. The fields that need to be *dupped* are pointers, where the reasonable action is to heap allocate a new instance and assign over the data, dynamic arrays and associative arrays. Regarding pointers, this creation scheme will not work for all situations because with pointers you are dealing not only with the referenced item but also the identity. That is, pointers are used for indirection and allow for things like cycles and a strategy of following pointers and recreating them can lead to infinite loops. Here is the outline of a mixin for support.

```
const string PostBlit = `
  this(this) {
    alias typeof(this) T;
    foreach (i, field ; typeof(T.tupleof)) {
      alias typeof(T.tupleof[i]) FieldType;
      static if(isPointer) {
        ... Allocate new and assign
      } else static if(isDynamicArray!(FieldType)) {
        ... dup fields
      } else static if(isAssociativeArray!(FieldType)) {
        ... dup values, reuse keys
      } else {
        pragma(msg, "Unhandled␣type␣in␣postblit␣", T.tupleof[i]);
        assert(false);
      }
    }
  }
`;
```

With this new postblit, the offending `struct A` can now look like:

```
struct A {
  mixin(PostBlit);
  mixin(OpEquals);
  int[] int_a;
}
```

The nice thing about a mixin that automatically examines all the fields and provides suitable action is that when new fields are added there is a safety net. This can eliminate a lot of tedious code.

3 Automating More Functionality With Mixin

3.1 OpCmp

Suppose you want your *struct* to be able to be stored as a key in an associative array or in a `RedBlackTree`. Both of these require a suitable `opCmp`. Storing your *struct* as a key in an associative array has additional requirements discussed later.

`opCmp` provides for ordered comparison and in `D` calls to `<` and `>` are lowered into calls to `opCmp`. Often times you want your data ordered, but you don't care so much what the order is as long as its consistent. In fact this is the generic requirement of many containers (i.e. strict weak ordering). This function is simple to provide via mixin since the logic is straightforward: Given two *structs* to compare, start comparing the fields of each in turn, as soon as one is found to be less than or greater than the other, return the result. Again, this screams recursion because comparisons of types lead to comparisons of types.

With a suitable `typesDeepCmp` a mixin for `opCmp` can be defined as:

```
int opCmp(const ref typeof(this) other) const {
  return typesDeepCmp(this, other);
}

int opCmp(const typeof(this) other) const {
  return typesDeepCmp(this, other);
}
```

As previously mentioned the `<` gets lowered into a call to `opCmp`. Specifically, `a<b` becomes `a.opCmp(b)<0`. The function `a.opCmp(b)` is required to return a number less than 0 if `a<b`, a number greater than 0 if `a>b` and 0 if they are equal. Essentially, the requirements of `opCmp` are **more** than those of `<`, yet `<` gets lowered to a call to `opCmp`. In terms of performance, this sounds like a recipe for disaster. Consider the following inefficient implementation:

```
import std.stdio;
bool isLessThan(string s1, string s2) {
  writeln("is␣'", s1, "'␣<␣'", s2, '"');
```

```
  return s1<s2;
}
struct MyString {
  int opCmp(const ref MyString other) {
    if(isLessThan(s,other.s)) {
      return -1;
    } else if(isLessThan(other.s,s)) {
      return 1;
    } else {
      return 0;
    }
  }
  private string s;
}
void main() {
  MyString m1 = { "this␣is␣a␣big␣string" };
  MyString m2 = { "this␣is␣a␣big␣string" };
  writeln(m1>m2);
}
```

The purpose of the `isLessThan` is just to catch the call and do a print statement. Replace the calls of `isLessThan(a,b)` with `a<b` and the same problem exists. The issue is the objects have been compared twice, to determine if one is less than the other. The fix is to replace the `if...else if...else` entirely with a call to a suitable `opCmp` if it exists. For strings, that call is simply `cmp`, so after importing `std.string` the more efficient implementation becomes:

```
struct MyString {
  int opCmp(const ref MyString other) {
    return s.cmp(other.s);
  }
  private string s;
}
```

In this case the type string has a `cmp` function already, so using it provides the efficiency. This generalizes to other types. So, identifying the pattern, a refinement to the description becomes: Given two *structs* to compare, start comparing the fields of each in turn using brute force if necessary but preferring field type's `opCmp` if it exists. As soon as one is found to be less than or greater than the other, return the result. This type of refinement can be achieved with the call to `__traits(compiles,...)`. This nice feature gives you the ability to see if for a specific type you can successfully do something, otherwise find a better way. In this example it allows us to see if we can compile the call to `opCmp`, and if so we use it, otherwise we compare on our own.

```
int opCmpPreferred(T, F)(const ref T lhs, const ref F rhs) {
  static if(__traits(compiles, (lhs.opCmp(rhs)))) {
    return lhs.opCmp(rhs);
  } else {
    return typesDeepCmp(lhs, rhs);
  }
}

int typesDeepCmp(T,F)(auto ref T lhs, auto ref F rhs) if(is(Unqual!T == Unqual!F)) {
  static if(isFloatingPoint!(T)) {
    ...
  } else static if(isSomeString!T) {
    return lhs.cmp(rhs);
  } else static if(is(T == struct)) {
      foreach (i, ignore ; typeof(T.tupleof)) {
        int result = opCmpPreferred(lhs.tupleof[i], rhs.tupleof[i]);
        if(result) return result;
      }
      return 0;
  } else static if(isPointer!(T)) {
    ...
  } else static if(isAssociativeArray!(T)) {
    ...
```

```
  } else {
    return (lhs < rhs)? -1 : (lhs > rhs)? 1 : 0;
  }
}
```

3.2 Hashing

Providing support for hashing, like comparison, can be quite burdensome. In D, for a given *struct* to support hashing it needs three things: (1) `toHash`, (2) `opEquals`, and (3) `opCmp`. `C++` newcomers to `D` often need to make an adjustment when storing data in associative arrays. In `C++` the common approach to creating an associative array mapping type `Key` to type `Value` is to provide an `operator<` for the key, then typedef a map:

```
typedef std::map<Key,Value> Map;
```

With this you are good to go. Perhaps you are using something that is less efficient when it comes to finding values, since you are not using a hash map - but you easily have the required tool.

Shift to `D` and things are a bit different because the default associative array does not have *deep copy semantics* and is a hash map. How do you easily get your associative array in D? At this time (`dmd` now at v2.06), a general map is conspicuously missing from the standard container library (`Phobos`). They do have `RedBlackTree`, the algorithm which is the basis of `std::map`, but they don't have the convenient wrapper `map` class. So if you are a `C++` developer used to composing with `std::map` and maybe you have never dealt much with hashing, you have a bit of extra work.

You could write a very simple `Map` template struct that makes use of `RedBlackTree` on your own and then you would feel at home. But think of how may times you've read in `C++` books how lame it is to write your own containers. Leave it to the professionals is the advice - and it is good advice. (Personally I would love to see in `D` something like `std.cpp.Map`, `std.cpp.Vector`, `std.cpp.Deque`, etc just to make the transition easier).

Alternatively, you could bite the bullet and start using the built-in associative array as your primary source for such types. To do this you need to provide hash functions. Briefly, think of hashing like organizing your sock drawer. Initially the drawer is just a big pile of sock pairs (assuming the clothes folder of the house always takes the time to bundle socks that go together into a ball before throwing them into the drawer). When you need a good pair of gray socks you open the drawer and start to search. After years of fumbling in the dark you come up with a great idea - divide the drawer into sections, one for white, one for gray, one for black and one for everything else. You've just created a hash, with the hashing function of identifying one of { white, gray, black, other }. How good the hashing function is depends very much on the data. If you are a goth, a gloomy sort or in corporate America you might have many more pairs of black socks than the others. If you want to look for a specific pair of black socks you still may have some digging to do. Without special knowledge of what is searched most often, it becomes clear that the goal of your hashing should be to distribute the socks among the dividers in the drawer equally.

Can this be done well generally? I don't know, but I've heard it said that given any hash function there is a set of data that can bring it to a crawl. So there are risks in terms of performance. Keep in mind, though, that all hash functions can simply `return 0;` and they **will** work, although the performance degrades to a linear search. Contrast this issue of poor performance with one of incorrect behavior that occurs when an `opCmp` or `opEquals` does not take into account the deep semantic nature of the object. In one case the search can take a very long time. In the other the search comes up empty when the item is really there. So, generally, any hashing that focuses on the actual data (not pointers) is better than none.

It is a straightforward task to create a mixin that incorporates all data into the hash function. The field types requiring special attention are those with *reference semantics*, which are pointers, associative arrays, dynamic arrays, and user defined types with *reference semantics*. Because recursion can be employed, the issue with UDTs with *reference semantics* dissolves into recursive calls that deal with the issue. As long as you have a mechanism that can deterministically combine hash-codes of composed objects you can build up a custom hash code. One common approach is to add new hash information as follows:

```
new_hash_code = current_hash_code*prime_number + field_hash_code;
```

With this approach there is no guarantee you have the best hashing function, but it should be good for most uses.

```
const string ToHash = `
  /** Hashing function hitting all data - mileage may vary. */
  hash_t toHash() const /* pure */ nothrow {
    return fieldsToHash!(typeof(this))(this);
  }
`;

hash_t deepHash(T)(const ref T t) nothrow {
  const int prime = 23;
  hash_t result = 17;
  static if(isPointer!T) {
    if(t) { result = result*prime + deepHash(*t); }
  } else static if(isSomeString!(T)) {
    result = result*prime + typeid(T).getHash(&t);
  } else static if(isAssociativeArray!T) {
    try {
      foreach(key, value; t) {
        result = result*prime + deepHash(key);
        result = result*prime + deepHash(value);
      }
    } catch(Exception) {
      assert(0);
    }
  } else static if(isArray!T) {
    ...
  } else static if (is(T == struct) || is(T == class)) {
    ...
  } else {
    assert(false);
  }
  return result;
}
```

As with the `typesDeepCmp` we can create a similar function `toHashPreferred` to ensure that any custom hash functions along the way are preferred over our definition.

```
hash_t toHashPreferred(T)(const ref T t) nothrow {
  static if(is(T==struct) && __traits(compiles, (t.toHash()))) {
    return t.toHash();
  } else {
    return deepHash(t);
  }
}
```

With the new mixin, turning a *struct* into a hash-able key is simple. Since we need three new functions pulled, the *HashSupport* mixin does this.

```
const string HashSupport = `
  mixin(OpEquals);
  mixin(OpCmp);
  mixin(ToHash);
`;
```

Now using `mixin(HashSupport)` gives the ability to turn a basic *struct* into something store-able in the built-in associative array. The following example demonstrates this - the output shows that the associative array is now able to find the person by hashing on the `Key`. It prints `Found PersonData(3)`.

```
import std.stdio;
import std.datetime;
import opmix.mix;
struct Key {
  mixin(HashSupport);
```

```
  string _name;
  Date _birthDate;
}
struct PersonData {
  int _favoriteNumber;
}

void main() {
  PersonData[Key] personMap;
  personMap[Key("John␣Doe", Date(1960, 1, 15))] = PersonData(3);
  // .... later and it is time to lookup
  auto key = Key("John␣Doe".idup, Date(1960, 1, 15));
  if(auto found = key in personMap) {
    writeln("Found␣", *found);
  } else {
    assert(0);
  }
}
```

3.3 Assignment

So, what about assignment. Now that we have `mixin(PostBlit)` and `mixin(OpEquals)`, a similar `mixin(OpAssign)` should be cake. The question is, is it necessary? Consider this example:

```
import std.stdio;
import std.traits;
import opmix.mix;
struct S {
  mixin(PostBlit);
  mixin(OpEquals);
  char[] mutable;
}
void main() {
  S s1 = {['a','b'].dup};
  S s2;
  s2 = s1;
  writeln(s1.mutable.ptr, "␣", s2.mutable.ptr);
  writeln(s1==s2);
}
```

Providing output:

```
7F23EA945FE0 7F23EA945FD0
true
```

In the above example, you might expect a problem without an `opAssign`. We have just assigned an `s1` to `s2` and effectively gotten a deep copy, even though we have not implemented `opAssign`. How did that happen?

If you come from `C++` there would be understandable confusion. Effectively we have a `postblit` taking the place of our `C++` copy constructor and we have `opEquals` taking the place of our `operator==`. Now, here we have a *reference cluct* called mutable. Granted, many types (vectors, lists, strings) in `C++` already have the *deep copy semantics* built it. But assume we have a field in `C++` where that is not the case. For example, in terms of semantics a `char[] mutable` member in `D` is similar to a `int *_i` member in `C++` in that both require extra work to get back deep semantics. In `C++`, though, we learned that we needed not only a copy constructor and equality comparison overload, we also needed an assignment operator overload (`operator=`). In this `C++` example all three are provided and the results show that `s1` and `s2` are equal and not sharing data, so nice *deep copy semantics*. If the `operator=` is removed `s1` and `s2` will be equal because they will share data.

```
#include <iostream>
struct S {
  int *_i;
  S(int i) {
    _i = new int;
```

```
    *_i = i;
  }
  S(const S& other) {
    if(other._i) {
      _i = new int;
      *_i = *other._i;
    }
  }
  bool operator==(const S& rhs) const {
    return ((this==&rhs) or (_i and rhs._i and (*_i == *rhs._i)));
  }
  S& operator=(const S& other) {
    if(other._i) {
      _i = new int;
      *_i = *other._i;
    }
  }
};

int main(int argc, char** argv) {
  S s1(1), s2(1);
  s1._i = new int;
  *s1._i = 42;
  s2 = s1;
  std::cout << "Are␣sharing␣" << (s1._i == s2._i) <<
    "␣are␣equal␣" << (s1 == s2) << std::endl;
  return 0;
}
```

But there is a big difference between `D` and `C++` when it comes to assignment. So, what happened in our example above is `D` provided a default `opAssign` that calls the `postblit` automatically. That is, if your `postblit` already provides deep copy semantics, you get deep copy semantics on assignment for free. But there is still a bit of a hitch when it comes to `const` and `immutable` types.

3.4 *dup* and Global Dup

Why do some types have a *dup* property and others not? Likely the reason is some are better served living with *reference semantics* but allowing the user to deep copy with *dup*. Other reasons might be if the class deals directly with the heap, uses pointers, or manages its own memory. Dynamic arrays (including string), associative arrays, some containers (`SList`, `DList`, `Array`,...), `BitArray`, `CodepointSet` are all examples of structs containing *dup*.

If you are not dealing with any of those issues you may not need a *dup* type function. But, we can still provide our own global, non-member, *dup* function, say *gdup* and it may have a use with the *copy of const* issue.

3.4.1 Copy of Const

How do you copy a const object? This depends very much on if the object is a *reference cluct* or *value cluct*. For example, this works fine:

```
import std.stdio;
struct S {
  string s;
}
void main() {
  const(S) s = {"test"};
  S other = s;
  writeln(other, "␣and␣", s, "␣are␣the␣same␣", other==s);
}
```

However, a very similar example without the magic of storing `immutable(T)[]` leads to a problem.

```
import std.stdio;
struct S {
  char[] c;
```

```
}
void main() {
  const(S) s = {"test"};
  S other = s;
}
```

In this example we get a compile error:

```
(11): Error: cannot implicitly convert expression (s) of type const(S) to S
```

This highlights the issue that it can be very difficult to get a copy of a `const` or `immutable` object in `D` when there are *reference semantics* involved. The reason hinges on `D`'s concept of *transitive const*, which means `const` in `D` ascribes the same `const` nature not only to the object, but all objects in its composition recursively. In this example, one component of `S`, specifically `c` has reference semantics when it comes to its *non-const* elements. As it stands, if the language did allow you to blindly copy the `const(S)` item into a `non-const` item it would break the *transitive const* guarantee because the newly minted `other` which is `non-const` could update the contents of `c` which is really shared. This is the same type of sharing that we overcame with `mixin(PostBlit)` and that served us well by giving us a chance to make our own copies. But the compiler does not make us implement the `postblit`. We could have just as easily left it out and lived with the sharing. Alternatively, we could have implemented `postblit` and only *dupped* some fields and not others, allowing some sharing. Given that the compiler can not know, after leaving the `postblit` whether the newly minted item is in fact a true deep copy of all composed objects, recursively, it can not allow us to use that function to copy a `const(T)` object.

What you really want is a function that can take an instance, mutable or immutable, and duplicate it in a way that guarantees no sharing. With this function copying `const` or `immutable` objects becomes a non-issue.

```
import std.stdio;
import opmix.mix;
struct S {
  mixin(OpEquals);
  char[] c;
}
void main() {
  const(S) s = {"test"};
  S other;
  gdup(other, s);
  writeln(s.c.ptr, "␣and␣", other.c.ptr,
          "␣are␣different␣but␣they␣compare␣",
          s==other);
}
```

Effectively, `gdup` has allowed us to break the sharing and get a true deep copy, without the `postblit`. There are implications here in terms of performance. One of the primary reasons the language kept structs simpler and used the `blit` with user opt-in for `postblit` is the performance benefit of just copying contiguous data around. It is fast. For the call to `gdup` we are forced to create a new empty object and it is the job of `gdup` to fill in the fields doing the same recursively.

- Pain Point for C++ Programmers

  The issue of how to get mutable objects from `const` objects in `D` is still being hammered out. There are many threads in the news groups about the best approach and the potential problems with the `postblit`. But most of the time that comes from wanting to get copies of objects that you have claimed are `const` only because you want the code to work for both `const` and non-const values. A perfect example is with this idiom in `C++`. Assume you are developing a `C++ class` and it has a large member called `BigObject` which has been created with nice deep copy semantics. You know anytime it is going to be passed around it should be by reference because it is big. Now, having read all about encapsulation and the benefits of hiding data, you decide to provide a `const&` read accessor and a `const&` accepting write accessor because both are necessary for your use case. So, I know some of you are thinking - "why would you do that"? If you have a private member for which you give all rights for the user to muck with then there is no point in encapsulating it,

just make it public because effectively that is what it is. Well in C++ there is a point, because you may want to do something extra on the write of the field, like grab a lock to synchronize access; and forcing them to go through a method call is an indirection providing the encapsulation which gives you the leeway to change how you see fit in the future.

```
class X
{
public:
  BigObject const& big_object() const {
    return big_object_;
  }

  void big_object(BigObject const& val) {
    big_object_ = val;
  }

private:
  BigObject big_object_;
};
```

The problem with transporting this type of logic to D is that the assignment of `val`, which is `const` into `big_object_` which is non-const is forbidden in D, without a suitable `opAssign` taking a `const` type, if that `BigObject` is a *reference cluct*.

So, here is one use case that apparently calls for creating a non-const deep copy of a `const` object. Evaluating further, though, with D we have properties. And in D the argument of "why would you do that" is pretty sound; you would not, at least not initially without the need for the special logic like locking. In D you can make a field private and provide no access to it. But, you can also make it **public** and still get the leeway you had in C++ when employing both read and write accessors. Later you can decide to make that public field private, by changing its name and then providing the read/write accessors to satisfy any existing usage of `object.field` in the wild. In the write accessor you can add the needed special logic, say for locking.

So, starting out public without read/write accessors makes sense in D when read/write access is required. Now move forward in time to the day you make your app multithreaded and decide it is time to add the write accessor for locking. Well you are back to the same issue - you can not get around it. If your type is a *reference cluct* taking it as a `const` or `const ref` parameter is fine as long as you don't touch it. But it is very hard for a C++ developer to think that copying or assigning a `const` into a non-const is in any way modifying the data. But it is - and the reason it is is that while the act itself does not make any change to the original `const` object, future actions on the non-const copy can and likely will.

- Where do `const` objects come from?

  Here are two interesting threads in the newsgroup that highlight the issues with `postblit` and copying of `const(T)` when `T` is a *reference cluct*.

  - http://www.digitalmars.com/d/archives/digitalmars/D/Copying_structs_with_pointers_140951.html
  - http://www.digitalmars.com/d/archives/digitalmars/D/learn/Confused_about_const_19185.html

  When you look at the roadblocks people are hitting when dealing with parameters of `const(T)` and `ref const(T)` it is pretty clear that a global *dup*, if it can be done preserving all `const` and `immutable` guarantees is what is needed. I think *bearophile* says as much in this quote:

  > In practice to copy something const to something that's not const you need a deep copy function, because inside the array there can be other arrays that are const, etc. Transitive const requires transitive copy, it's easy :-)

  This is what the *gdup* does.

3.4.2 *dup* Mixin

Now that we have a global *dup* function - *gdup*, we can copy objects of type `const(T)`:

```
void workWithBig(const ref Big big) {
  Big bigCopy;
  gdup(bigCopy, big);
}
```

But this is a bit ungainly. A more convenient feeling syntax would be:

```
void workWithBig(const ref Big big) {
  Big bigCopy = big.dup;
}
```

And this can be achieved with `mixin(Dup)`. In this case the `const(Big)` is first deep copied with *dup* and then `postblit` into `other`.

```
import std.stdio;
import std.traits;
import opmix.mix;
struct Big {
  mixin(PostBlit);
  mixin(OpEquals);
  mixin(Dup);
  char[] c;
}
void main() {
  const(Big) s = {"test"};
  Big other = s.dup;
  writeln("char␣data␣", s.c.ptr, "␣and␣", other.c.ptr,
          "␣are␣different␣(i.e.␣sans␣sharing)␣but␣Bigs␣compare␣",
          s==other);
}
```

A downside to this approach is it changes the established convention somewhat. Before, *dup* was kind of reserved for *reference clucts*, i.e. types retaining *reference semantics* by not going the extra mile for *deep semantics*. In this usage, though, we give a *dup* to `Big` which has gone the extra mile to gain *deep semantics* via the `mixin(PostBlit)` and `mixin(OpEquals)`. But the intended usage here is to provide an ability to copy `const(T)` in a convenient way, so it makes sense.

4 Constructors

From TDPL: "The presence of at least one constructor disables all of the field-oriented constructors", and "the compiler always defines the no-arguments constructor".

`MORE TO COME`

5 Conventions with members

5.1 Properties

In `C++` there are no properties. To achieve encapsulation it is common to make a member variable private. Then provide a *read* accessor for *read only* variables and both a *read* and *write* accessor for *read write* variables. This helped because if a change in the implementation were desired (e.g. read/write from file, db, cache etc) it could be changed in the accessor methods without affecting client code.

D, with properties is an improvement. Assume these three classifications, from most to least encapsulating:

- *inaccessible*
- *read only*

- *read write*

To achieve *inaccessible* simply make the member private.

When developing a *struct* start with all members *inaccessible* and only change if there is a real need to give up the access. In D, make the field you might in the future want accessed as `foo.name` be called `_name` and declare it private. So an `int` field named age would be `private int _age`.

For *read only* access provide a getter property named `name`.

```
import opmix.mix;
import std.stdio;
struct PersonalInfo {
  pure public auto @property age() const { return _age; }
  private {
    int _age;
  }
}
void main() {
  const(PersonalInfo) elderly = {92};
  writeln("The␣elderly␣person␣is␣", elderly.age);
}
```

This is pretty clear, but it is not as clear as:

```
import opmix.mix;
import std.stdio;
struct PersonalInfo {
  mixin(ReadOnly!_age);
  private {
    int _age;
  }
}
void main() {
  const(PersonalInfo) elderly = {92};
  writeln("The␣elderly␣person␣is␣", elderly.age);
}
```

There is no need for a boilerplate *read write* access to a private member. If *read write* access is required for a member, name the member `name` and make it public. In the future you can change your mind and control how the clients access the data. You just can not control if they do, so once you decide to go *read write* assume everyone is touching your junk.