Have you ever tried using immutable structs in D? Have you ever succeeded?

Taming immutable with librebindable
• Refresher:
• We write software following Domain-Driven Design.
  - "Treat the value object as immutable. Make all operations side-effect-free functions that don’t depend on any mutable state.
  ...
  Domain events are ordinarily immutable, as they are a record of something in the past."
    – Domain-Driven Design Reference
• Data being manipulated by domain code should never be mutated!
  - Instead, always return new data.
  - Makes code testable, predictable, avoids spooky action at a distance.
  - Heavy use of UFCS chains of ranges to keep memory usage limited.
• If possible, we want to guarantee this statically.
Taming immutable with librebindable

We want to use immutable.

• Our traditional code:

```cpp
struct ArrayContainer {
    @ConstRead
    private int[] array_
;
    ... mixin(GenerateFieldAccessors);
}
```

• template based:
  - slow to build, high memory usage
  - brittle, sensitive to compiler changes

• nontransitive: potential copy of array_
  needed

• What we would like:

```cpp
immutable struct ArrayContainer {
    int[] array;
}
```

• simple
• fast
• readable
• cheap invariant checks: only in the constructor
• transitive: no dup on get
Taming immutable with librebindable

But we can’t use immutable!

- We want to use immutable data types.
- We want to use ranges.
- We also use a lot of associative arrays.
- But immutable data types break many ranges, and don’t work with associative arrays!

```cpp
immutable struct S { int i; } 
...
auto list = [S(2), S(4), S(3)];
assert(list.maxElement!“a.i“ == S(4));
```

Error: cannot modify struct instance `extremeElement` of type `S` because it contains `const` or `immutable` members
Taming immutable with librebindable

But we can’t use immutable!

```rust
immutable struct S { int i; }
...
auto list = [S(2), S(4), S(3)];
assert(list.maxElement!“a.i“ == S(4);
```

Error: cannot modify struct instance `extremeElement` of type `S` because it contains `const` or `immutable` members

- This keeps happening, and it will keep happening.
- Why?
- It is extremely intuitive that this should work:

```rust
T fun(T)(T arg1, T arg2) {
    T result = arg1;
    result = arg2;
    return result;
}
```
Taming immutable with librebindable

The solution is not Unqual!

• Maybe we can just do this?

```c
T fun(T)(T foo, T bar) { Unqual!T result = foo; result = bar; return result; }
```

• Unqual strips immutable from immutable struct, but the fields are still immutable!

• This was a surprise to me:

```c
immutable struct S { int i; }
void main() {
  Unqual!S s;
  static assert(!is(typeof(s) == S));
  s = S(5);
}
```

• Error: cannot modify struct instance `s` of type `S` because it contains `const` or `immutable` members

• s.i is still immutable int! The Unqual, it does nothing!
Taming immutable with librebindable

What actually goes wrong if you overwrite immutable?

- Important to understand the actual danger!
- The problem is not mutating an immutable field.
- The problem is observing an immutable reference change its value.

```cpp
immutable struct S {
    int field;
}

void genericFun(T)(T first, T second) {
    auto store = first;
    immutable int* ptr = &store.field;
    int firstField = *ptr;
    store = second; // danger!
    int secondField = *ptr;
    // This fails!
    assert(firstField == secondField);
    // We have observed an immutable pointer change its value.
    // All is lost, etc.
}
```
Possibilities that we discarded

- Maybe we can relax the const system with headmut?
  - We cannot wait for possible future features!
- Can we do headmut in a library?
- First attempt: Turducken types.
- Our type T, packed in a struct, packed in a union.¹
- Because it’s a struct, we’re allowed to use `std.algorithm.mutation.moveEmplace()` even though T has immutable members
- Because it’s a union, the destructor is not called!

```rust
struct Turducken(T) {
    Turkey store;

    struct Turkey {
        Duck duck;
    }

    union Duck {
        Chicken chicken;
    }

    alias Chicken = T;
}
```

¹ Thanks @n8sh!
Taming immutable with librebindable

Turducken Types

- Because it’s a struct, we’re allowed to use `std.algorithm.mutation.moveEmplace()` even though T has immutable members
- Because it’s a union, the destructor is not called!
  - We can control the lifetime!
  - Yes, this is intentional!
  - Why do we need to control the lifetime? headmut: we want to overwrite an already stored value.
- Downsides: founded on quicksand, undefined features, and bugs.
- Can break any day, or if you look at it wrong.
- Not a good solution.

```cpp
struct Turducken(T) {
    Turkey store;
    struct Turkey {
        Duck duck;
    }
    union Duck {
        Chicken chicken;
    }
    alias Chicken = T;
}
```
How to write a headmut

- What’s the simplest thing that could work?
  ```c
  align(T.alignof)
  struct HeadMut(T) {
    void[T.sizeof] data;
  }
  ```

- GC issue with precise scan:
  - void[n] is always treated as pointers.

- What we want is a type "like T, but mutable": DeepUnqual!
Taming immutable with librebindable
DeepUnqual to the rescue!

- rebindable.DeepUnqual defines equivalent types for every D data type.
- Mutable pointers in the same place, mutable nonpointers in the same place.
- Otherwise, nothing in common.

| struct {} | { DeepUnqual!member for each member } |
| union, K[V] | void[T.sizeof] |
| class, interface, function, T* | void* |
| T[] | { length, ptr } |
| delegate | { void*, void* } |
| basic type T | Unqual!T |
Taming immutable with librebindable

DeepUnqual to the rescue!

- DeepUnqual!T will be scanned by a precise GC at exactly the fields that T will be scanned
- Same size
- Same alignment (chaotic neutral)
- Working with it is deeply unsafe.
- You have to cast everything.

```java
Rebindable!T

void set(T);

T get():

T goes in, T comes out never a miscommunication
```
Taming immutable with librebindable

DeepUnqual to the rescue!

- How do I store a T in it? How do I get a T back out?
  - rebindable.Rebindable wrapper!
    - Rebindable value;
    - value.set(S(5));
    - assert(value.get == S(5));
    - Even if S is immutable.

But is this safe?

void set(T);
T get():

T goes in, T comes out
never a miscommunication
Taming immutable with librebindable

Is it secret? Is it safe?

• It’s safe precisely so long as T is secret.
• D immutable mashes together memory immutability and observability.
• We don’t care if a value changes, so long as we can never observe it changing.
• Rebindable!T is a boxed T.
• It is never possible to get a reference to the contained data in T form, because get returns by value.
• The stored data is mutable thanks to DeepUnqual, so mutating it is arguably correct: Rebindable!T never mutates memory that was not declared as mutable.
• And the memory itself is not exposed as immutable, but get returns an immutable value copy.
Taming immutable with librebindable

Is it secret? Is it safe?

• The stored data is mutable thanks to DeepUnqual, so mutating it is arguably correct: Rebindable!T never mutates memory that was not declared as mutable.
• And the memory itself is not exposed as immutable, but get returns an immutable value copy.
• While the data is stored in Rebindable, it can change, but every change must be from a valid state to a valid state, and we can never catch it in mid-change.
• Weakness: T cannot rely on the fact that every address it lives at was created by a constructor or copy constructor call.
  • If the copy constructor monkeys around with field addresses, it will break.
Taming immutable with librebindable

What can we do with this?

- librebindable ships with:
- immutable-safe Nullable, rebindable.Nullable:
  ```c
  Nullable!(const int) ni;
  assert(ni.isNull);
  ni = 5;
  assert(!ni.isNull && ni.get == 5);
  ni.nullify;
  assert(ni.isNull);
  ```

- immutable-safe associative arrays, rebindable.AssocArray:
  ```c
  AssocArray!(int, S) assocArray;
  assocArray[0] = S([5]);
  assocArray[0] = S([6]);
  assert(assocArray[0] == S([6]));
  ```
Taming immutable with librebindable

Proposal: Referenceability is the &root of all evil.

- Is immutable too strong? Should we be able to overwrite immutable fields?
- No: immutable is too weak!
- The problem is that we can observe immutable fields changing.
- By default, we can just do &i and get a permanent view at i: &i is immutable(int)*, but may change value!
- What’s stronger than immutable? rvalue!
- For the non-compiler developers: An assignment has the form:
  - left value = right value;
  - So "lvalue" = "anything that can appear on the left of an assignment operator."
  - And "rvalue" = "anything that can only appear on the right side of an assignment operator."
Taming immutable with librebindable

Proposal: Referenceability is the &root of all evil.

- lvalue = rvalue;
- A hypothetical rvalue struct {} can be stronger than immutable and still be overwritten by assigning a new value!
- rvalue struct is a pure data struct: no taking the address of fields, no referencing fields, no assigning fields directly.
- Why? We need not fear mutation because nobody can catch us in the act.
- And if we only assign newly constructed values of T, we never break invariant either.
- A pure, rvalue variable is stronger than immutable: it is unreferenceable, and hence, unobservable.
- In fact, it is memoryless: it cannot be thought of as "data stored at an address in memory", but only as "data in itself".
- Its value may be read, but its fields are not remotely observable – because they are Plain Old Values.
Taming immutable with librebindable

Actually viable proposal: Outright remove immutable struct fields.

- Why doesn’t Unqual!T work? Why shouldn’t Unqual!T work?
- Lots of code, even in Phobos, already assumes that Unqual!T makes a head-mutable T.
- Structs can have immutable fields hidden inside, creating isolated patches of immutability.
- Unqual already creates head-mutable values for every type other than structs.
- Instead we should do this:
  - fields are always actually head-mutable, immutable(T) only sets the default for T var;

```plaintext
immutable struct S {
    // actually immutable(int)[] a
    // but can only be accessed // via immutable this
    int[] a;
}
```
Taming `immutable` with `librebindable`

Actually viable proposal: Outright remove `immutable` struct fields.

- In effect, `immutable` struct makes each field `Unqual!(immutable T)`.
- But: direct field access `(T.field)` on non-`immutable T` implconvs `T` to `immutable first`
  - If this is impossible, just error.
- This preserves constness guarantees, preserves invariants, no accessors needed
- But allows usage of any `T` in data structures via `Unqual!T`.
- **Effect: Unqual!T == HeadMutable!T, always.**
- Doesn’t fully solve the headmut problem for classes. `(immutable(Object) will never implconv to Object.)`
  - But we don’t use classes in domain code anyways. :-)
  - Anyways, `std.typecons.Rebindable` exists.

```haskell
immutable struct S
{
  // actually immutable(int)[] a
  // but can only be accessed
  // via immutable this
  int[] a;
}
```
Taming immutable with librebindable

But for now, librebindable will do.

- Package is available at https://code.dlang.org/packages/rebindable
- Code is hosted on https://github.com/FeepingCreature/rebindable/
- Tested and working in production code.
- Thanks to my employer, Funkwerk, for letting me work on this!
- Questions?