Mutability Wildcards in D

Steven Schveighoffer
http://schveiguy.com/dconf2016.pdf

Mutability

By default, variables in D are *mutable*

Mutable variables can be modified in place.

```
int x = 0;
int *p = &x;      // pointer to x
x = 1;            // set the value of x
assert(*p == 1); // setting value does not make a copy.
```

Immutable Data

Immutable data in D is *guaranteed* by the compiler against modification.

Designated with the `immutable` type modifier

```
immutable(int) x = 0; // never mutable!
x = 1;                // compiler error!
```

A type modifier tells the compiler to *modify* how it treats the type inside the parentheses.

Tail Modifiers

Type modifiers apply to parentheses only.

```
immutable(int) x, y;
immutable(int *) px = &x; // immutable pointer to immutable data
immutable(int)* mpx = &x; // mutable pointer to immutable data
px = &y; // Error
mpx = &y; // OK
```

Tail modifiers only modify “target” data type.

Transitivity

Every composed member inside a modified type is also modified.

```
struct S {
    int x;
    int *y;
}

immutable(S) s;
static assert(is(typeof(s.x) == immutable(int)));
static assert(is(typeof(*s.y) == immutable(int)));
```

This means `immutable` data cannot refer to mutable data.

```
int x = 5;
immutable(int)* y = &x; // Error!
```

The `const` Wildcard

`const` indicates that data cannot be modified through *that reference*.

`const` can refer to both mutable and `immutable` data.

```
int x = 5;
immutable(int) y = 6;
const(int) *z = &y;
z = &x;
x = 7;
assert(*z == 7);
```

`const` Function Parameters

Arguments passed to `const` parameters cannot be modified inside that function via that parameter

```
void foo(const(int *) p) {
    *p = 5; // Error
}
```

Can accept mutable, `immutable`, or `const` data references.

Why `const`?

Function signature advertises guarantee, no need to read the function body. Good for component development.

Const always available as a wildcard at any point, not just at function calls.

Compiler enforces guarantee.

One function implementation emitted by the compiler.

Virtual functions allowed. Allows base class to enforce API on derived classes.

Where `const` fails

Once you go `const` you cannot go back.

```
struct S {
    private int         _x;
    ref int             x()           { return _x; }
    ref const(int)      x() const     { return _x; }
    ref immutable(int) x() immutable { return _x; }
}
```

Double Indirection

Mutable and `immutable` cannot convert to `const` under two indirections of mutable pointers.

```
immutable(int) ix;
const(int) *ip = &ix; // OK
int *p;
const(int) **pp = &p; // Error
*pp = ip;             // Because now p points at ix
*p = 5;               // mutates ix, illegal!
```

The Template Solution

Templates can alleviate some problems for const

```
struct S {
    private int _x;
    ref x(this T)() { return _x; }
}

S            mx;
const(S)     cx;
immutable(S) ix;
static assert(is(typeof(mx.x()) ==           int  ));
static assert(is(typeof(cx.x()) ==     const(int) ));
static assert(is(typeof(ix.x()) == immutable(int) ));
```

Has same code generation problems as 3-function solution.

But it is shorter and DRYer than the previous boilerplate.

The `inout` Wildcard

`inout` *wraps* a mutability modifier upon a function call. Then it *unwraps* the modifier after the function returns.

This allows it to transfer the mutability modifier from the **in**put of the function to its **out**put.

```
struct S {
    private int _x;
    ref inout(int) x() inout { return _x; }
}
```

`inout` Rules

Must act as a stand-in for only *one* mutability modifier.

Wrapping occurs only on function calls. This is to ensure the one-at-a-time property holds, and compiler can reason with opaque functions.

After wrapping, no other mutability modifiers can implicitly convert to `inout`, and `inout` cannot implicitly convert to any other modifiers except `const`

Compiler prevents mutation, and `inout` is transitive.

Globals or static function variables cannot be `inout`

`inout` and Double Indirection

The double indirection rule *does not apply* to `inout`.

```
void foo(inout(int)** p, inout(int)* t) {
    *p = t;
}

int x;
int *p;
foo(&p, &x); // OK!
immutable(int) ix;
immutable(int) *ip;
foo(&ip, &ix); // OK!
```

Calling `inout` with Different Modifiers

When two or more `inout` parameters are called with differing type modifiers, a common modifier is used.

```
inout(int)* minp(inout(int)* p1, inout(int)* p2) {
    return *p1 < *p2 ? p1 : p2;
}

int x;
const(int) cx;
immutable(int) ix;

static assert(is(typeof(minp(&x,  &cx)) == const(int) *));
static assert(is(typeof(minp(&x,  &ix)) == const(int) *));
static assert(is(typeof(minp(&cx, &ix)) == const(int) *));

static assert(is(typeof(minp(&x,  &x )) ==           int  *));
static assert(is(typeof(minp(&ix, &ix)) == immutable(int) *));
```

Calling `inout` with `inout`

`inout` functions can be called with `inout` as the type modifier.

```
inout(int)* minp(inout(int)* p1, inout(int)* p2) {
    return *p1 < *p2 ? p1 : p2;
}

inout(int)* minp3(inout(int)* p1, inout(int)* p2, inout(int)*p3) {
    inout(int)* r1 = minp(p1, p2);
    return minp(r1, p3);
}
```

In the calls to `minp`, `inout` wraps `inout`, and unwraps to `inout`.

`inout(const)` Modifier

`inout(const(int))` can only mean:

```
static assert(is(/*mutable*/const(int) ==     const(int)));
static assert(is(     const(const(int)) ==     const(int)));
static assert(is(immutable(const(int)) == immutable(int)));
```

Because of this, `immutable` can implicitly convert to `inout(const)`.

`inout` functions called with both `immutable` and `inout` will wrap and unwrap as `inout(const)`.

Why `inout`?

Does not suffer from double indirection problem.

Useful in situations where result needs same mutability modifier as parameters. e.g. accessor.

`inout` still only emits one function binary. `inout` is *not* a template

Like `const`, function signature advertises the guarantees.

Compiler enforces guarantee. Same as `const`.

Virtual functions allowed. Allows base class to enforce API on derived classes.

Double Indirection Use Case

```
void popFront(T)(ref T[] a) @safe pure nothrow @nogc {
    a = a[1 .. $];
}
```

Modified to use `inout`:

```
void popFront(T)(ref inout(T)[] a) @safe pure nothrow @nogc {
    a = a[1 .. $];
}
```

New version advertises that `popFront` will not modify array data.

It also will generate less code.

`const` and `inout` are Viral

If you are writing templates or functions with `inout` or `const`, then the type parameters you use must support `inout` or `const` as well.

```
struct S {
    private int          _x;
    ref int              x()            { return _x; }
    ref const(int)       x() const      { return _x; }
    ref immutable(int) x() immutable { return _x; }
}

auto ref foo(T)(auto ref inout(T) t) { return t.x; }
```

`inout` and Nested Functions

inout nested functions can access the inout data in the context frame.

```
immutable(int) b;
inout(int)* foo(inout(int)* x) {
    inout(int)* fun(inout(int)* y) {
        return *x < *y ? x : y; // access to x allowed
    }
    version(none) return fun(&b); // could violate immutable
    return fun(x);  // OK.
}
```

`inout` and Delegates/ Function Pointers

If a delegate parameter contains `inout` parameters, those are *independent* from the wrapping of the enclosing function

```
inout(int)* foo(inout(int)* function(inout(int)*) f,
                inout(int)*x) {
    return f(x);
}

void main() {
    int x;
    foo((int *a) { *a += 1; return a; }, &x); // error
}
```

`inout` Member Variables

`inout` is not allowed as a member of a struct or class.

Wrapping/unwrapping of a struct member would not be possible.

```
struct S {
    inout(int) x; // error
    int y;
}
```

`inout` Without `inout` Parameters

Compiler disallows tagging a return as `inout` when no parameter is tagged as `inout`.

Local variables cannot be `inout` if no parameter is tagged as `inout`.

Constructing `inout` Data

Only `inout` modified types can be constructed with `inout` data.

```
    struct S {
        int *x;
        this(int *a) {x = a;}
    }

    void foo(inout(int)* x) {
        auto s = S(x); // Error
        auto s2 = inout(S)(x); // Error
    }
```

In order to create an `inout` structure with a defined constructor, you must tag the constructor as `inout`

```
        this(inout(int) *a) inout {x = a;}
    ...

    void foo(inout(int)* x, int *y) {
        auto s = inout(S)(x); // Now OK
        auto s2 = S(y); // Works with inout ctor
    }
```

Fixing `inout`

1. Allow local variables that are `inout`, which can only be constructed as new data.

2. Allowing `inout` return type for non-`inout` functions.

Allow `inout` Members

If we allow structs/classes with `inout` members, we can restrict them to cases where wrapping/unwrapping does not occur.

```
struct S {
    inout(int)* x;
}

inout(int)* foo(S s) {
    return s.x;
}

inout(int)* bar(inout(int) *x) {
    auto s = S(x);
    return foo(s);
}

void baz() {
    int x = 1;
    int *p = foo(S(&x));
}
```

Allow Temporary Unwrapping

Potentially allow parameters of delegates/function pointers to participate in wrapping/unwrapping

```
inout(int)* foo(inout(int)* function(inout(int)*) f,
                inout(int)*x) {
    return f(x);
}

void main() {
    int x;
    foo((int *a) { *a += 1; return a; }, &x); // Could allow
}
```

Compiler still guarantees foo code cannot modify directly. Only delegates/function pointers that have the same mutability modifier could be wrapped.

Questions?