Copy and Move Semantics

in the

D Programming Language

Ali Çehreli

DConf 2013

Inroduction

- Basic concepts
  - User-defined types
  - Fundamental operations
  - Immutability
- A search of idioms and guidelines

Idioms of other languages may not be applicable

Two C++ guidelines:

- "Make everything **`const`** until you can't."
- "Pass objects by reference if they are expensive to copy."

```
// C++
MyInt average(const MyInt & lhs, const MyInt & rhs);
const MyInt result = average(var, MyInt(1));

const_taking(result);
```

May not be applicable in D:

```
// D
MyInt average(ref const(MyInt) lhs, ref const(MyInt) rhs)
const result = average(var, MyInt(1));    // ← compilation ERROR

// May fail in the future due to a change in MyInt
immutable_taking(result);

// May fail in the future
immutable imm = var;
```

Some definitions

- Type semantics
  - Value semantics
  - Reference semantics
- Kinds of values
  - lvalue (*left-hand* side value)
  - rvalue (*right-hand* side value)
- Type qualifiers
  - *mutable*
  - **`immutable`**
  - **`const`**

  (***`shared`*** *is out of the scope of this presentation.*)

Value semantics versus reference semantics

Easy to distinguish by the behavior of the `=` operator.

- **Value semantics**: Variables represent separate states

```
a = b;
assert(a == b);
a.mutate();
assert(a != b);    // separate objects
```

```
       a                 b
---+-------+---   ---+-------+---
   | state |         | state |
---+-------+---   ---+-------+---
```

- **Reference semantics**: Variables are handles to the same state

```
a = b;
assert(a == b);
a.mutate();
assert(a == b);    // two handles to the same object
```

```
                            a                 b
---+-------+---      ---+-------+---   ---+-------+---
   | state |            |   .   |         |   .   |
---+-------+---      ---+---|---+---   ---+---|---+---
       ▲                    |                 |
       |                    |                 |
       +--------------------+-----------------+
```

Lvalues versus rvalues

Lvalues

- can be on both sides of an assignment operation
- can have addresses
- can be bound to a reference

Simple example: "Named variables are lvalues."

Rvalues

- cannot be on the left-hand side of an assignment operation
- cannot have addresses
- cannot be bound to a reference

Simple example: "Literals and temporary variables are rvalues."

```
int foo() { return 1; }    // foo's return value is an rvalue
void bar(ref const(int) i) {}
// ...
    foo() = 2;             // ← compilation ERROR
    int * p = &(foo());    // ← compilation ERROR
    bar(foo());            // ← compilation ERROR
```

Rvalues cannot be bound even to const references

```
struct S
{}

void foo(ref const(S) p) {
    /* ... */
}

// ...

    foo(S());    // ← compilation ERROR
```

See http://wiki.dlang.org/DIPs and http://forum.dlang.org/ for frequent improvement requests for allowing this.

Type qualifiers

There are three kinds of mutability:

- *Mutable*: The state can be mutated (the default; no keyword)
- **`immutable`**: The state never mutates
- **`const`**: The state is not mutated through this reference

Then there is the wildcard:

- **`inout`**: Placeholder for the previous three (compiled as **`const`**).

Significant: Both **`immutable`** and **`const`** are transitive. The entire state that is reachable through a variable is also **`immutable`** or **`const`**, respectively.

User-defined types in various languages

Language	`struct`	`class`
C	value	-
C++	value	value
Java	-	reference
C#	value	reference
D	value	reference

`struct` versus `class` in D

D structs are somewhere between C structs and C++ structs.

`struct`

- Value type
- Scoped lifetime, allowing the RAII idiom
- No OOP
- Layout and alignment control of members
- *more...*

`class`

- Reference type
- Garbage collected
- Supports OOP
- *more...*

Fundamental object operations

Construction

- From scratch
- As a copy of another object
- By moving from an rvalue

Mutation (optional)

- Incrementally
- As a whole
  - By assigning from another object (copy new state + destroy old state)
  - By swapping with an rvalue

Destruction (not always)

D support for fundamental operations

Operation	**`struct`**	**`class`**
Construct from scratch	automatic	automatic
Construct as a copy	automatic	user-defined
Construct by moving from an *rvalue*	automatic	N/A
Mutate incrementally	automatic	automatic
Mutate as a whole by assigning from an *lvalue*	automatic	user-defined
Mutate as a whole by swapping with an *rvalue*	automatic	N/A
Destroy	automatic*	automatic*

* *Depends on whether the object is scoped or dynamic;*
and if dynamic, whether the runtime has decided to destroy it.

N/A: Class objects are never rvalues.
Not their values, but their handles appear in expressions.

Default class semantics

No state mutation.

- **Copy syntax**: A class variable starts its life as an additional reference to an existing garbage-collected class object.

```
auto b = a;          // b starts life as another handle
                     // to a's object
assert(b is a);
```

- **Assignment syntax**: A class variable disassociates from its object and becomes a handle to another object.

```
a = c;               // a is now another handle
                     // to c's object (not b's anymore)
assert(a is c);
```

User-defined `class` semantics

One possibility:

```
class C {
    // ...

    inout(C) dup() inout {
        // ... make a copy ...
        return new inout(C)(/* ... */);
    }

    // (Not a common operation.)
    void takeOver(C rhs) {
        // ... move the state of rhs to this object ...
    }
}
```

```
    auto m = new C;
    auto i = new immutable(C);
    auto c = new const(C);

    auto m_dup = m.dup();
    auto i_dup = i.dup();
    auto c_dup = c.dup();

    // inout produces correct types
    static assert(is(typeof(m_dup) == C));
    static assert(is(typeof(i_dup) == immutable(C)));
    static assert(is(typeof(c_dup) == const(C)));

    m.takeOver(m_dup);
```

Copying a `struct` object
The postblit function

`struct` objects are copied automatically by the following algorithm:

1. Bit-copy from source to destination (aka *blit* (bit level transfer))

   (This is so fundamental that self-referencing structs are not valid in D.)

2. If defined, execute the *post-blit* function of the type (presumably to make corrections to the copied object)

```
struct S {
    int[] data;

    this(this) {
        data = data.dup;
    }

    // ...
}
```

`struct` semantics for *lvalue* on the right-hand side

- **Copy syntax**: An object starts its life as a copy of an existing object.

```
auto b = a;    // copy: 'b' starts life as a copy of 'a'
assert(b is a);
```

- **Assignment syntax**: An object becomes a copy of another object. The old state on the left-hand side gets destroyed.

```
b = c;     /* copy + destroy: 'b' is now a copy of 'c';
                               the old 'b' is destroyed */
assert(b is c);
```

The assignment algorithm for *lvalue* on the right-hand side

```
    auto src = S();
    dst = src;    // right-hand side is an lvalue
```

The algorithm is efficient and exception safe:

```
// Equivalent pseudo-code
{
    // Make an actual copy of src (maybe expensive and may throw)
    auto tmp ←deep-copy← src;

    auto dst ⇦bit-swap⇨ tmp;

} // 'tmp' destroys the old state here
```

This is an improvement over C++, where the default behavior of assignment does not have the strong exception safety guarantee.

`struct` semantics for *rvalue* on the right-hand side

```
// The return value is an rvalue
S foo() { /* ... */ }
```

- **Copy syntax**: An object starts its life by a *bit-copy* of the rvalue and the rvalue's destruction is elided.

```
     auto a = foo();  /* move: 'a' starts life with the state
                                  of the returned object */
```

- **Assignment syntax**: The two states are effectively swapped.

```
     a = foo();       /* swap: 'a' takes over the state of the
                                 right-hand side object */
```

The assignment algorithm for *rvalue* on the right-hand side

```
    dst = foo();    // right-hand side is an rvalue
```

```
// Equivalent pseudo-code
{
    dst ⇦bit-swap⇨ rvalue;

} // 'rvalue' destroys the old state here
```

`immutable` values

```
    immutable i = 42;
    immutable s = S(1);
```

- Deep guarantee: Any state that is accessible through this variable is **`immutable`** as well.
- Bonus: Is implicitly **`shared`** (no need to lock in multi-threaded code).

Can be copied from *mutable* and **`const`**:

```
    auto m = 42;
    immutable im = m;    // automatic copy from mutable int

    const c = 43;
    immutable ic = c;    // automatic copy from const(int)
```

This slide is too optimistic because there is no mutable indirection *here.*

`const` values

```
const c = 42;
const s = S(1);
```

- Deep guarantee: No state that is accessible through this variable can be modified
- (*no compatibility with* ***`shared`***)

Can be copied from *mutable* and **`immutable`**:

```
auto m = 42;
const cm = m;         // automatic copy from mutable int

immutable i = 43;
const ci = i;         // automatic copy from immutable(int)
```

const versus immutable values

```
void foo_byValue(int i)           { /* ... */ }
void foo_i(ref immutable(int) i) { /* ... */ }
void foo_c(ref const(int) i)     { /* ... */ }

// ...

    immutable i = 1;
    foo_byValue(i);
    foo_i(i);
    foo_c(i);

    const c = 1;
    foo_byValue(c);
    foo_i(c);    // ← compilation ERROR
    foo_c(c);
```

So, is a **`const`** value less useful than an **`immutable`** value?

Guideline 1 (deceptive!)

Observation: **`const`** values cannot be passed to functions taking reference to **`immutable`**.

Deceptive guideline: "If a variable is never mutated, make it **`immutable`**, not **`const`**."

const references

```
class C {    // reference type
    // ...
}
```

Inclusive: Can refer to *mutable*, **`immutable`**, and **`const`**.

```
    auto m = new C;
    const(C) c = m;
    static assert(is (typeof(c) == const(C)));    // *
```

```
    auto i = new immutable(C);
    const(C) c = i;
    static assert(is (typeof(c) == const(C)));    // *
```

```
    auto c_ = new const(C);
    const(C) c = c_;
    static assert(is (typeof(c) == const(C)));    // *
```

* The actual type qualifier has been lost on **`c`**: It is always **`const(C)`** regardless of the actual object that it refers to.

`const` reference parameters

Message to the caller:

"I shall not mutate your argument."

```
class C { /* ... */ }

void foo(const(C) p) {
    // ...
}
```

Accepts *mutable*, **`immutable`**, and **`const`**.

```
    auto m = new C;
    auto i = new immutable(C);
    auto c = new const(C);

    foo(m);
    foo(i);
    foo(c);
```

The actual type qualifier has been lost on **`p`** *inside the function.*

immutable references

Exclusive: Can refer to only **`immutable`**.

```
immutable(C) i0 = new C;                // ← compilation ERROR
immutable(C) i1 = new immutable(C);
immutable(C) i2 = new const(C);         // ← compilation ERROR
```

immutable reference parameters

Message to the caller:

"I shall not mutate your argument but you must not mutate it either."

```
void foo(immutable(C) c) {
    // ...
}
```

Accepts only **`immutable`**:

```
    auto m = new C;
    auto i = new immutable(C);
    auto c = new const(C);

    foo(m);    // ← compilation ERROR
    foo(i);
    foo(c);    // ← compilation ERROR
```

Guideline 2 (deceptive!)

Observation: **`const`** reference parameters are inclusive and **`immutable`** ones are exclusive.

Deceptive guideline: "If a reference parameter is not going to be mutated by the function, make it **`const`**, not **`immutable`**."

```
void prettyPrint(const(char)[] str) { /* ... */ }

void main()
{
    char[] m;
    string i;          // ('string' is the same as immutable(char)[])
    const(char)[] c;

    prettyPrint(m);
    prettyPrint(i);
    prettyPrint(c);
}
```

`immutable` reference would be limiting:

```
// Would not accept char[] or const(char)[]
void prettyPrint(string str) { /* ... */ }
```

Guideline 2 is deceptive (1)

Unfortunately, **`const`** erases the actual type qualifier.

When the function needs to pass the parameter to an **`immutable`** reference, it must make a copy it:

```
import std.conv;

void usefulFunction(string str) { /* ... */ }

void prettyPrint(const(char)[] str) {
    // ...
    usefulFunction(str);                // ← compilation ERROR
    usefulFunction(to!string(str));     /* ← compiles but sometimes
                                             the copy is unnecessary */
}
```

A template solution is wordy and may increase the size of the program:

```
import std.conv;
import std.traits;

void prettyPrint(T)(T str)
    if (isSomeString!T)
{
    // ...
    usefulFunction(to!string(str));  // no-op if already immutable
}
```

Guideline 2 is deceptive (2)

Programming convenience brings runtime cost:

```
struct Archiver {
    string fileName;

    this(const(char)[] fileName) {
        this.fileName = fileName.idup; /* unnecessary if the
                                          arg is already immutable */
    }

    ~this() {
        // ... use this.fileName ...
    }
}
```

```
    char[] m;
    string i;
    const(char)[] c;

    // Convenient:
    Archiver(m);
    Archiver(i);    // unnecessarily expensive
    Archiver(c);
```

Guideline 2 is deceptive (2)
(a compromise)

Take reference to **`immutable`**:

```
struct Archiver {
    string fileName;

    this(string fileName) {
        this.fileName = fileName;    // no copy necessary
    }

    ~this() {
        // ... use this.fileName ...
    }
}

// ...

    char[] m;
    string i;
    const(char)[] c;

    Archiver(m.idup);    // copied by the caller
    Archiver(i);         // no cost
    Archiver(c.idup);    // copied by the caller
```

- A worry: Some information leaks out. (Note that reference to **`const`** does not have this issue.)
- No big deal: In D, the callee asks a favor from the caller: "I need an **`immutable`** anyway; please make a copy yourself if you have to."

Guideline 2 (revised)

"Make the parameter reference to **`immutable`** if that is how you will use it anyway. It is fine to ask a favor from the caller."

Guideline 1 (again)

Deceptive guideline: "If a variable is never mutated, make it **`immutable`**, not **`const`**."

```
struct MyInt {
    int i;
}

void main() {
    auto m = MyInt(42);
    immutable i = m;        // so far so good
}
```

Let's imagine that the library adds a mutable indirection in the future:

```
struct MyInt {
    int i;
    private int[] history;
    // ...
}

void main() {
    auto m = MyInt(42);
    immutable i = m;        // ← compilation ERROR
}
```

So, perhaps **`const`** is better after all:

```
    const i = m;        // now compiles
```

Guideline 1 (revised)

"If a variable is never mutated, make it **`const`**, not **`immutable`**."

Will it work with functions that take **`immutable`** reference?

Options:

- If safe, efficiently convert **`const`** references to **`immutable`** by **`assumeUnique`** (no copy made):

```
void foo(immutable(MyInt)[] p) { /* ... */ }

// ...

    const(MyInt)[] c;
    c ~= MyInt(42);

    auto i = assumeUnique(c);
    foo(i);
    assert(c is null);    // at the expense of losing 'c'
```

- If not safe, make an **`immutable`** copy and pass it to the function.
- (Avoid!) If safe, reach for **`cast`** momentarily (no copy made):

```
    foo(cast(immutable(MyInt)[])c);
```

Return mutable value (guideline 3)

"Return mutable if the returned value is actually mutable."

Why prevent the caller from mutating a freshly made mutable result?

```
dstring foo() {
    dstring result;
    result ~= 'a';
    return result;
}

// ...

    auto s = foo();
    s[0] = 'A';          // ← compilation ERROR
```

Returning mutable would be more useful:

```
dchar[] foo() {
    dchar[] result;
    result ~= 'a';
    return result;
}

// ...

    auto s = foo();
    s[0] = 'A';          // now compiles
```

Return value being used as immutable

On the other hand, a mutable result would be inconvenient if the caller needed **immutable** to begin with:

```
dchar[] foo() { /* ... */ }

// ...

    dstring imm = foo();    // ← compilation ERROR
```

Options:

- The return value of a **pure** function can be implicitly convertible to **immutable**:

```
pure dchar[] foo() { /* ... */ }

// ...

    dstring imm = foo();    // now compiles
```

- Document that calling **assumeUnique** on the result would be safe:

```
/* This function returns a unique string. */
dchar[] foo() { /* ... */ }

// ...

    auto m = foo();
    immutable i = assumeUnique(m);
    assert(m is null);
```

Construction syntax

Which construction syntax to use?

```
immutable s0 = S(42);          // type qualifier
```

Type qualifiers can be used as *type constructors* to "build a new type from an existing one". The following line has a subtle semantic difference from the previous one:

```
auto s1 = immutable(S)(42);    // type constructor
```

Guideline 4

"Prefer the *type constructor* syntax."

Tools

Here are some tools that help with defining a **`struct`**:

```
struct S {
    int[] data;

    this(string s) {
        data.length = 42;
    }

    this(this) {
        // post-blit to make a correction. e.g.
        data = data.dup;
    }

    this(S rhs) {
        // 'rhs' is a copy of the argument; do move...
    }

    this(ref const(S) rhs) {
        // 'rhs' is an lvalue; do copy...
    }

    ref S opAssign(S rhs)     {
        // 'rhs' is a copy of the argument; swap with this...
        return this;
    }

    ref S opAssign(ref const(S) rhs) {
        // 'rhs' is an lvalue; copy to this and destroy old state ...
        return this;
    }
}
```

Summary

We would like to have simple guidelines that help with day-to-day programming.

Here are a few:

1. If a variable is never mutated, make it **`const`**, not **`immutable`**.
2. Make the parameter *reference to* ***`immutable`*** if that is how you will use it anyway. It is fine to ask a favor from the caller.
3. Prefer returning mutable reference if the state is mutable to begin with.
4. Prefer type constructor syntax to type qualifier syntax.