Binding Rvalues to `ref` Parameters

Prepared for DConf 2019

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

May 10, 2019

Credits

- *Manu Evans:* raised the issue, authored DIP 1016
- *Walter Bright:* specification, implementation details

Motivation

- Two reasons for `ref` in function signatures:
  1. Function wants to manipulate a parameter
  2. Function wants to take/return a large object efficiently

- Problem: the language only caters for (1)

Efficient call/return protocol

- Often better to traffic large/elaborate objects by pointer
- Using actual pointers clunky and unsafe
- Often efficiency is a primary concern, not side effects
- Yet the language is worried that side effects will not last

Example

```
struct Point {
  long x, y, z, color;
  ...
}
Point p;
Point origin();
double distance(ref const Point, ref const Point);
...
// desired: auto n = distance(p, origin());
auto t = origin();
auto n = distance(p, t);
```

Workaround: Overloading FTW!

```
// using struct Point defined above
double distance(ref const Point p1, ref const Point p2)
{ ... implementation ... }
double distance(const Point p1, ref const Point p2)
{ return distance(p1, p2); }
double distance(ref const Point p1, const Point p2)
{ return distance(p1, p2); }
double distance(const Point p1, const Point p2)
{ return distance(p1, p2); }
```

- Scales with 2^n, oi!

Related Work

- Binding rvalues to `const T&` fundamental in C++
- So tight, you can't overload on l/rvalues
- Part of the motivation for `T&&`

- Rust can bind rvalues to ref with syntax on the caller side, e.g. `fun(&mut 42)`

So let's relax the rule!

`ref` shall accept rvalues!

Obvious Issue

- Adapted from [Stroustrup D&E]

```
void bump(ref long x) { ++x; }
...
int counter;
bump(counter);
```

- `int` to `long` implicit conversion
- If this compiled, `counter` would be unmodified!
- Fragility, too: changing types in code that works!

Too Much Binding? No Problem!

- New rule!
- “Rvalues bind to `ref`, EXCEPT when they originate from lvalues by means of implicit conversion.”
- Introducing exceptions is worrisome. . .

. . . And Indeed. Consider:

```
struct Widget {
  public double price;
  ...
}
void applyDiscount(ref double p) {
  p *= 0.9;
}
...
Widget w;
w.price = 100;
w.applyDiscount;
assert(w.price == 90);
```

Make It a Property

```
struct Widget {
  private double _price;
  double price() { return price; }
  void price(double p) { assert(p > 0); price = p; }
  ...
}
...
Widget w;
w.price = 100;
w.applyDiscount;
assert(w.price == 90); // oops
```

But Wait, There's More

- Functions and nonmember properties

```
int x = global; // variable or function call
global = 42; // variable or function call
fun(global); // will this change global or not?
```

- Even worse with indexing operators

```
Tensor t;
t[0] = 42; // ref or opIndexAssign
t[0] += 7; // ref or opIndexOpAssign
fun(t[0]); // will this change t[0] or not?
```

- All generic code will need to mind this

The Problem

- Fundamentally, identical syntactic forms differ radically in semantics
  - Caller passes a modifiable expression, e.g. `t[1]`
  - Callee changes its parameter per the contract
  - Both play “nice” but protocol fools both
- Surprising bugs
- Fragility in maintenance

Proposal

Plan

- Figure out matching rules
- Eliminate “bad” matches
- Devise code generation

Parameter matching rules (current)

- Four levels of matching params to args:
  1. no match
  2. match with implicit conversions
  3. match via qualifier conversion
  4. exact match
- Compute matching for each argument
- Take the *minimum* for the function

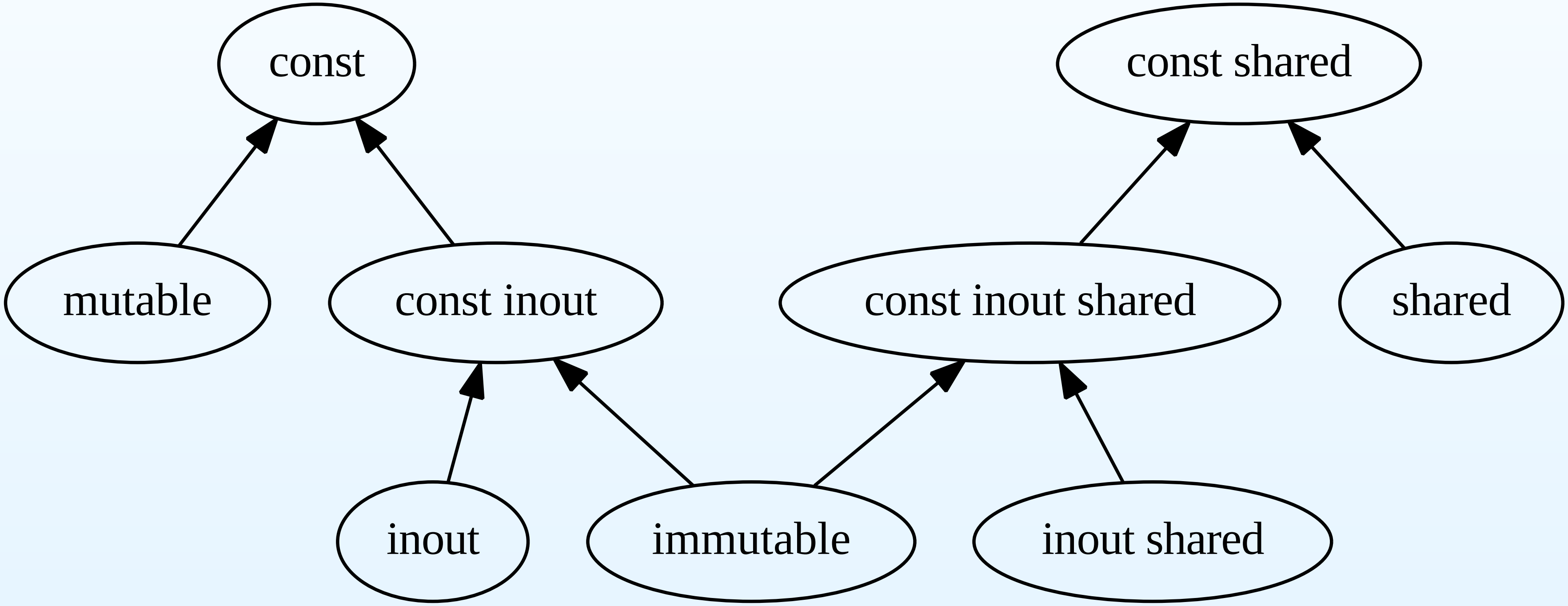
- Changing this list would be a major hurdle

Assignable Type & Expression

- Definition: We call a type T assignable iff T is unqualified or has the `shared` qualifier.
- Definition: We call an expression e assignable iff there exists some expression e_1 such that the syntactic form $(e) = (e_1)$ is a valid expression.

Recall Qualifier Conversion DAG

- Only modifiable quals are mutable and `shared`



Fork In The Road: Proposal 1

- To bind expression e of type `U` to `ref T`:
- If e assignable expression and `T` assignable type:
  - Return existing algorithm.
- Else run existing algorithm assuming e lvalue, get level x
  - If $x = 1$, return level 1 (no match).
  - Else return 2 (match via conversion).

Intuition

- Simple!
- Eliminate confusing cases of assignability
- Make binding to `ref` count as a conversion
  - No C++ mistake
  - Can still overload on `ref`

Aftermath

- Naturally eliminates a large class of bugs:

```
void bump(ref long x) { ++x; }
...
int counter;
bump(counter); // nope, assignable
bump(100L); // okay, level 2
bump(100); // okay, level 2
```

- Danger when both caller and callee wrongly believe mutation will occur

Overloading On `ref` Works. . .

```
void fun(ref int);
void fun(int);
fun(42); // level 2 vs level 4
int x = 42;
fun(x); // level 4 vs level 2
```

. . . With Quirks

```
void fun(ref int);
void fun(int);
const int x;
fun(x); // level 2 vs level 2, ambiguous
void gun(ref const int);
void gun(int);
const int gun();
fun(gun()); // level 2 vs level 2, ambiguous
```

Proposal 2

- No changes to parameter-level match!

Change Function-Level Matching!

- Run algorithm once assuming all lvalues, get all matches
- If one match, return it
- If more than one match, discard and defer to the *old* function-level algorithm

Aftermath

- Eliminates the confusing cases at argument matching level
- Backwards compatible
- `ref` and value interchangeable
- Complicated/clunky rules
  - Really adds a new matching level without adding one
- Slow (probably not a practical problem)

Code Generation

Gode Generation

- *Lifetime of temporaries* large part of proposal
- Intermingled with *order of evaluation*, too
- Problem: both were underspecified to start with
  - Also, quite complex
- DIP grew significantly

Solution

- Migrate order of evaluation to spec
- Migrate lifetime of temporaries to spec
- (Just document what’s there!)

DIP says

When binding to `ref` params, temporaries follow same rules as for binding to value params

Life, Simplified

- Huge simplification on all sides
  - Implementation
  - Understanding
  - Use
- Rules were complex to start with
  - "End of full expression except for the right-hand side of conditional expressions"
  - But. . . already implemented and in use

Lesson Learned: Proper Motivation is Key

- Motivation is the rocket fuel pushing the DIP forward
- “Chesterton’s Fence” essential, too
  - Understanding the situation allows for solutions

Lesson Learned: Integrate Within

- Language is underspecified
- A DIP sadly needs to fix some of the spec, too
- Sometimes need to read the actual implementation

- Key: improve spec first, build DIP on it!

Lesson Learned: Be Rigorous

- Approximate spec + approximate DIP = bad
- DIP should leave no room for interpretation
- The DIP will be implemented by a vengeful ex

Thank You!