Partially-Formed Objects
For Fun and Profit

Meeting C++ 2020 (virtual), 2020-11-14

Marc Mutz <marc.mutz@kdab.com>

Created on November 14, 2020

KDAB

The Qt, OpenGL and C++ Experts

In This Talk

- Partially-Formed
  - definition
  - examples from C/C++
- Moved-From Objects
  - `IndirectInt`
  - `std::remove_if`
  - safe and unsafe functions
- Composability
  - `flat_map`
- C++20 `std::movable<>`
- Bonus Slides:
  - Case Study: Pen
  - Weaker Move Semantics Models
  - Exception Guarantees
    - `std::variant`

Disclaimer

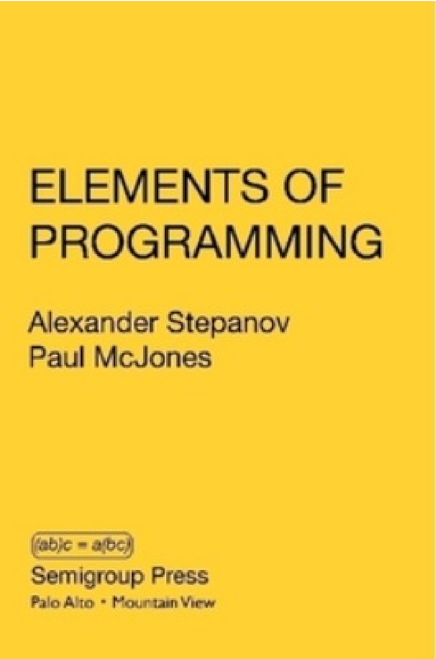
Disclaimer:

- all of the following is only for value types
  - Regular / SemiRegular
- not for RAII or polymorphic types

The Partially-Formed State in C/C++98/EoP

- **The Partially-Formed State in C/C++98/EoP**
- Move Semantics
- Composability
- C++20
- Bonus Slides

Elements Of Programming



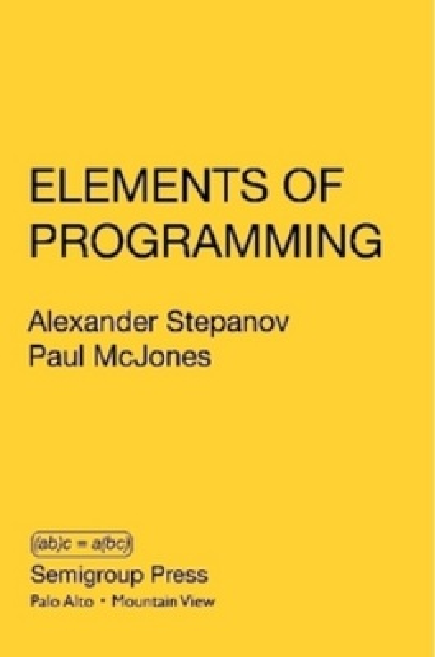
ELEMENTS OF PROGRAMMING

Alexander Stepanov
Paul McJones

(ab)c = a(bc)
Semigroup Press
Palo Alto • Mountain View

elementsofprogramming.com

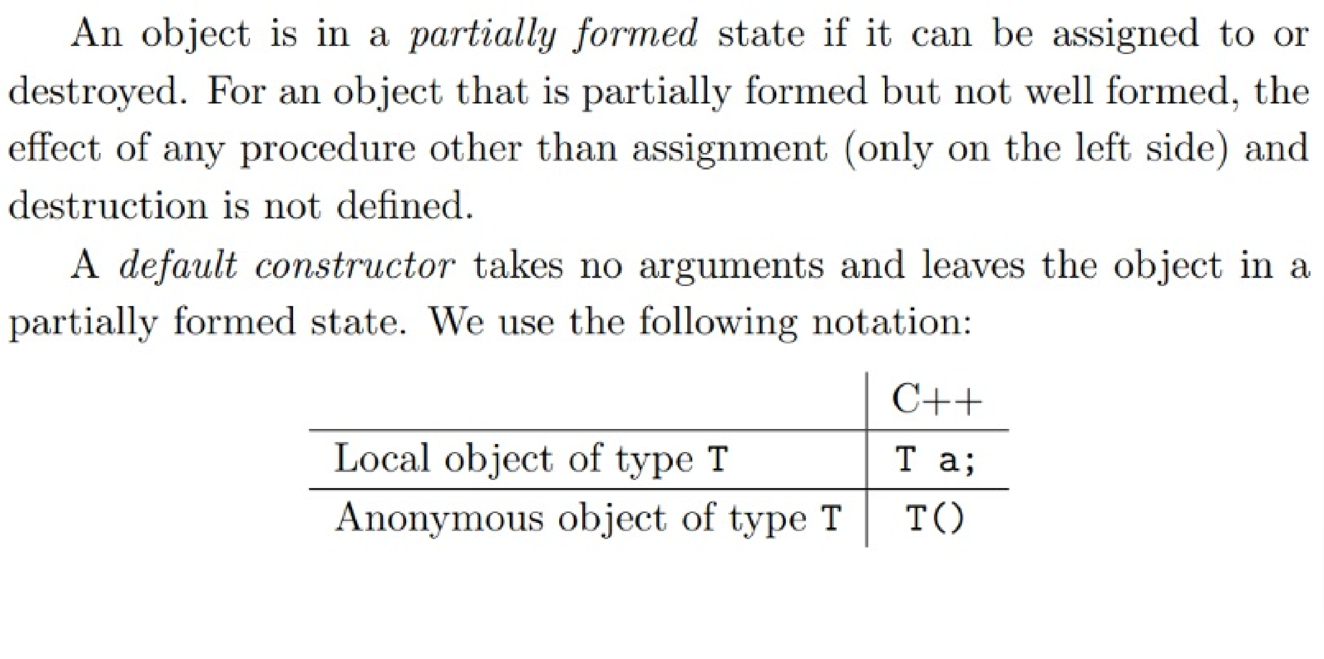
Elements Of Programming (cont'd)



ELEMENTS OF PROGRAMMING

Alexander Stepanov
Paul McJones

(ab)c = a(bc)
Semigroup Press
Palo Alto · Mountain View



An object is in a *partially formed* state if it can be assigned to or destroyed. For an object that is partially formed but not well formed, the effect of any procedure other than assignment (only on the left side) and destruction is not defined.

A *default constructor* takes no arguments and leaves the object in a partially formed state. We use the following notation:

	C++
Local object of type `T`	`T a;`
Anonymous object of type `T`	`T()`

elementsofprogramming.com

What's the Partially-Formed State?

An object is in the partially-formed state if it can be assigned-to and destroyed. (EoP)

Example 1.0

```
int i; // `i` is partially-formed
```

Example 1.1

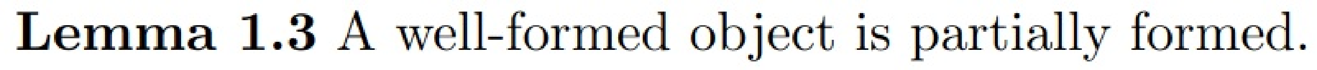
```
int i = 0; // is `i` is partially-formed?
```

What's the Partially-Formed State?

An object is in the partially-formed state if it can be assigned-to and destroyed. (EoP)

Definition of Partially-Formed

- clearly, every object is partially-formed
    - when assignable and destructable
- this is EoP Lemma 1.3:



Lemma 1.3 A well-formed object is partially formed.

- in this talk:
    - Partially-Formed := known to be partially-formed, not known to be well-formed

Example 1.1 (revised)

```
int i = 0; // `i` is partially-formed, not Partially-Formed
```

Example 2

```
std::string s; // `s` is partially-formed, not Partially-Formed
```

Example 3

Rect r; // `r` is Partially-Formed

Guideline 1

Unless you **know** that the type in question provides more, all you can **assume** is that a default-constructed object is Partially-Formed.

Partially-Formed is a **program state**, not a bit-pattern.

Example 4

```
auto p = new int{0};
delete p;
// `p` is ...?
```

What's the Partially-Formed State?

An object is in the partially-formed state if it can be assigned-to and destroyed. (EoP)

Example 4 (revisited)

```cpp
auto p = new int{0};
delete p;
// `p` is Partially-Formed
// C++: [basic.stc]/4 (footnote!)
```

Example 4.1

```cpp
auto p = new int{0};
auto q = p;
delete p;
// `p` and `q` are Partially-Formed
// cf. P1726 for an overview
```

Example 5

```
std::input_iterator auto it = ~~~;
auto jt = it;
++it;
```

What's the Partially-Formed State?

An object is in the partially-formed state if it can be assigned-to and destroyed. (EoP)

Example 5 (revisited)

```cpp
std::input_iterator auto it = ~~~;
auto jt = it;
++it;
// `jt` is Partially-Formed
// C++: [tab:inputiterator], EoP: Section 6.2
```

Example 5 (rewritten)

```
std::input_iterator auto it = ~~~;
auto jt = std::next(it);
auto value = *it;
```

Example 5 (rewritten) (cont'd)

```
std::input_iterator auto it = ~~~;
auto jt = std::next(it);            // UNSAFE function
auto value = *it; // oops, `it` was Partially-Formed
```

Example 5 (rewrite fixed)

```cpp
std::input_iterator auto it = ~~~;
std::advance(it, 1);                 // SAFE function
auto value = *it; // OK
```

Guideline 3

If you feel uncomfortable around Partially-Formed state, avoid it:

- Immediately-Invoked Lambda Expression (IILE)
- `std::unique_ptr`

But **not** in the type design!

- Default ctor need *not* establish a valid value.
  - "Don't pay for what you don't use"!
  - "When in Rome^WC++, do as the Romans^Wints do"!

Example 1.0 (fixed)

```cpp
int i = [&] {
    switch (~~~)
    case ~~~: return 42;
    ~~~~
    };
}();

use(i);
```

Example 1.0 (fixed wrongly)

```
int i = 0; // must ... always ... initialise

switch (~~~)
case ~~~: i = 42; break;
~~~~
};

use(i);
```

Example 4.1 (fixed)

```cpp
auto p = std::make_unqiue<int>(0);
auto q = p; // ERROR: move-only type
p.reset();
// `p` == nullptr --- well-formed
```

C/C++98/EoP Summary

- Be precise:
  - partially-formed = destructible and assignable
  - partially-formed-not-known-to-be-well-formed
  - partially-formed-known-not-to-be-well-formed
- Partially-Formed Objects exist in the language as early as K&R C / C++98:
  - *default-initialised* objects
  - invalid pointers
  - copies of an InputIterator since advanced
- Guidelines:
  - Unless you **know** that the type in question provides more, all you can **assume** is that a default-constructed object is Partially-Formed.
  - Partially-Formed is a **program state**, not a bit-pattern.
  - If you feel uncomfortable around Partially-Formed state, avoid it, but not in type design.

C/C++98/EoP Summary (cont'd)

References:

- Alex Stepnov et al (2009): http://elementsofprogramming.com
- Paul E. McKenney et al (2019): http://wg21.link/p1726
- std:
  - https://eel.is/c++draft/basic.stc#general-4
  - https://eel.is/c++draft/tab:inputiterator#row-6

Move Semantics

- The Partially-Formed State in C/C++98/EoP
- **Move Semantics**
- Composability
- C++20
- Bonus Slides

Example 6

Consider this C++98 class (by Geoffrey Romer):

```
class IndirectInt {
    boost::shared_ptr<int> m_i; // class invariant: never NULL
public:
    explicit IndirectInt(int i = 0) : m_i(boost::make_shared<int>(i)) {}
    // compiler-generated copy operations / dtor are ok!
    // (Rule Of Zero)

    friend bool operator==(const IndirectInt& lhs, const IndirectInt& rhs) {
        return *lhs.m_i == *rhs.m_i;
    }
    friend std::ostream& operator<<(std::ostream& s, const IndirectInt& i) {
        return s << *i.m_i;
    }
};
```

Herb Sutter (in "Move, Simply!"): `// Buggy class: Move leaves behind a null smart pointer`

"[A pure library implementation of move semantics] did not "automatically" move from rvalues which is a really nice feature of the current proposal. This allows **completely safe move semantics** to come into client code **with absolutely no code changes** for the client."

(N1377 (2002))

Guideline 4

Treat moved-from objects as Partially-Formed.

Guideline 4 (rephrased)

Unless you **know** that the type in question provides more, all you can **assume** is that a moved-from object is Partially-Formed.

Example 7

```
std::vector<IndirectInt> v = ~~~;
std::remove(v.begin(), v.end(), IndirectInt(0));
for (auto& i : v) // exposition only
    std::cout << i;
```

Guideline 5 (Very Old: Effective STL)

Treat objects in [std::remove, end) as Partially-Formed, use Erase-Remove-Idiom.

Example 7 (fixed)

```cpp
std::vector<IndirectInt> v = ~~~;
v.erase(std::remove(v.begin(), v.end(), IndirectInt(0)),
        v.end());
for (auto& i : v)
    std::cout << i;
```

Example 7 (fixed, [C++20])

```
std::vector<IndirectInt> v = ~~~;
std::erase(v, IndirectInt(0));
for (auto& i : v)
    std::cout << i;
```

Conjecture: Observable moved-from states

Conjecture:
The remove-like algorithms (`remove`/`remove_if`/`unique`) are the only cases where moved-from objects appear in a C++ program without an explicit cast (`std::move`, `std::forward`, `static_cast<T&&>`).

Guideline 6.1 (Sean Parent)

std::move() is an unsafe operation.

Guideline 6.2

std::move() is an unsafe operation.
But `std::exchange(., {})` is its *safe* companion.

Guideline 7

If you can't tolerate the thought of partially-formed objects,
prefer C++20 `erase_if()` over `std::remove_if()`,
and C++14 `std::exchange(x, {})` over `std::move(x)`.

Guideline 7'

If you can't tolerate the thought of partially-formed objects, prefer *safe* over *unsafe* functions.

Move Semantics Summary

- Valid C++98 programs become "invalid" C++11 ones unless you treat moved-from objects as partially-formed.
- Guidelines:
  - Treat moved-from objects as Partially-Formed.
  - Treat objects in [std::remove, end) as Partially-Formed, use Erase-Remove-Idiom, or `std::erase()`.
  - `std::move()` is an unsafe operation (and so are `std::remove()`, `std::remove_if()`, `std::unique()`).
  - If you can't tolerate the thought of partially-formed objects,
    - prefer *safe* over *unsafe* operations.
- References:
  - Scott Meyers (2001): Effective STL
  - Howard Hinnant (2002): http://wg21.link/N1377
  - Marc Mutz (2017): https://www.kdab.com/stepanov-regularity-partially-formed-objects-vs-c-value-types/
  - Herb Sutter (2019): https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf
  - Geoff Romer (2020): http://wg21.link/p2027
  - Herb Sutter (2020): https://herbsutter.com/2020/02/17/move-simply/

Composability

- The Partially-Formed State in C/C++98/EoP
- Move Semantics
- **Composability**
- C++20
- Bonus Slides

Partially-Formed States Are Closed Under Composition

```
struct VectorAndIndex {
    std::vector<int> vec;
    int index; // index into 'vec'
>;

```

Partially-Formed States Are Closed Under Composition

```
struct VectorAndIndex {
    std::vector<int> vec;
    int index; // index into 'vec'
>;

VectorAndIndex vi1;      // partially-formed-not-well-formed
```

Partially-Formed States Are Closed Under Composition

```
struct VectorAndIndex {
    std::vector<int> vec;
    int index; // index into 'vec'
>;

VectorAndIndex vi1;      // partially-formed-not-well-formed
VectorAndIndex vi2 = {}; // partially-formed-not-well-formed
```

Partially-Formed States Are Closed Under Composition

```
struct VectorAndIndex {
    std::vector<int> vec;
    int index; // index into 'vec'
>;

VectorAndIndex vi1;      // partially-formed-not-well-formed
VectorAndIndex vi2 = {}; // partially-formed-not-well-formed
auto vi3 = VectorAndIndex{{0, 1, 2}, 0};      // well-formed
auto vi4 = std::move(vi3);
              // vi3 is now partially-formed-not-well-formed
```

Lemma 1: Partially-Formed States Are Closed Under Composition

Lemma 1: Let $N>1$ and $\mathtt{T}_i$, $1 \leq i \leq N$, be semi-regular. Let `S` be a `struct { T₁ t₁; ...; Tₙ tₙ; }` and `s` ∈ *Domain*(`S`). Then $\exists 1 \leq i \leq N$: $\mathtt{t}_i$ Partially-Formed ⇒ `s` is Partially-Formed.

Proof: Trivial (by way of member-wise assignment and destruction).

std moved-from objects

Unless otherwise specified, moved-from objects are placed in a valid, but unspecified state.

std moved-from objects (cont'd)

valid, but unspecified <=>
can apply all wide-contract operations on the type

Lemma 2: valid-but-unspecified is *not* closed under composition

Lemma 2: valid-but-unspecified is *not* closed under composition.

Proof: By contradiction: Assume Lemma false, then `flat_map` doesn't need a custom move constructor.

Example 8

```
template <class K, class V, ~~~ Compare ~~~,
          typename KC = std::vector<K>,
          typename VC = std::vector<V>>
class flat_map {
    KC m_keys;
    VC m_values;
public:
    flat_map(flat_map&&) = default;
    auto size() const {
        return m_keys.size(); // P0429
        return m_values.size(); // ?
        return std::min(m_keys.size(), m_values.size()); // ?
    }
};
```

Example 8 (wrong fix)

```
template <class K, class V, ~~~ Compare ~~~,
          typename KC = std::vector<K>,
          typename VC = std::vector<V>>
class flat_map {
    KC m_keys;
    VC m_values;
public:
    flat_map(flat_map&& other) noexcept
        : m_keys(std::move(other.m_keys)),
          m_values(std::move(other.m_values))
    {   other.m_keys.clear();      // extra
        other.m_values.clear(); }  // work
    auto size() const {
        return m_keys.size(); // OK
    }
};
```

Composability Summary

Using language defaults...

- Partially-Formed objects composed are Partially-Formed
- Valid-But-Unspecified objects composed are *not* Valid-But-Unspecified

References:

- Zach Laine (2016..2019) https://wg21.link/P0429

C++20

- The Partially-Formed State in C/C++98/EoP
- Move Semantics
- Composability
- **C++20**
- Bonus Slides

std moved-from objects

Moved-from objects are placed in a valid, but unspecified state.

C++20 User-Type Constraining

- C++17 only promised this for std types
- C++20 seems to require this for user-types, too:
  - `std::movable` requires `std::move_constructible`
  - `std::move_constuctible` requires (in prose):
    - "[...] rv's resulting state [...] is valid but unspecified; [...]"

Guideline 8

Ignore [concept.moveconstructible]/1.3. It will be fixed.
No implementation can depend on it.

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`
- 2 Solutions:
  - Either make move-assign-from moved-from objects self-assign-save
    - Always the case when using the *Move-and-Swap Idiom*
  - Or provide `swap()` overload found using ADL that's self-swap-save
- None of these usually require extra work.

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

- 2 Solutions:

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

- 2 Solutions:
  - Either make move-assign-from moved-from objects self-assign-save

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

- 2 Solutions:
  - Either make move-assign-from moved-from objects self-assign-save
    - Always the case when using the *Move-and-Swap Idiom*

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

- 2 Solutions:
  - Either make move-assign-from moved-from objects self-assign-save
    - Always the case when using the *Move-and-Swap Idiom*
  - Or provide `swap()` overload found using ADL that's self-swap-save

The Hinnant Extension

- EoP's `swap()` works in terms of copies (like C++98 `std::swap()`)
- `std::movable` requires `std::swappable`
- In C++11+, `std::swap()` uses moves
- Now consider self-swap: `std::swap(x, x)`

```
// auto& lhs = x; auto& rhs = x;
T tmp = std::move(lhs);
lhs = std::move(rhs); // move-assigns-from moved-from object
rhs = std::move(tmp);
```

- 2 Solutions:
  - Either make move-assign-from moved-from objects self-assign-save
    - Always the case when using the *Move-and-Swap Idiom*
  - Or provide `swap()` overload found using ADL that's self-swap-save
- None of these usually require extra work.

Guideline 9

Prefer to provide an ADL `swap()` overload that is self-swap-safe.
Alternatively:
Ensure the move-assignment operator is self-swap-safe
even in the partially-formed state.

Guideline 10

Libraries may want to avoid depending on a working self-swap.

C++20 Summary

Guidelines:

- Ignore [concept.moveconstructible]/1.3. It will be fixed.
- Prefer to provide an ADL `swap()` overload that is self-swap-safe.
  - Alternatively, ensure the move-assignment operator is self-swap-safe
    - even in the partially-formed state.
- Libraries may want to avoid depending on a working self-swap.

References:

- https://eel.is/c++draft/concept.moveconstructible#1.3

End of Main Talk

Thank you for your attention!

Questions?

Bonus Slides

- The Partially-Formed State in C/C++98/EoP
- Move Semantics
- Composability
- C++20
- **Bonus Slides**
  - A Case Study: Pen
  - Weaker Models
  - Weak Exception Guarantees

Bonus Slides

- **A Case Study: Pen**
- Weaker Models
- Weak Exception Guarantees

A Case Study: Implicitly-Shared Pen I: SMFs

.h

```
class Pen {
    struct Private;
    Private *d; // Pimpl Pattern
public:
    constexpr Pen() noexcept
        : d{nullptr} {}
    Pen(const Pen& other);
    Pen(Pen&& other) noexcept
        : d{std::exchange(other.d, {})} {}
    Pen& operator=(const Pen& other)
    { Pen{other}.swap(*this); return *this; }
    Pen& operator=(Pen&& other) noexcept
    // Pen{std::move(other)}.swap(*this);
    { swap(other); return this; }
    ~Pen();
    void swap(Pen& other) noexcept
    { std::ranges::swap(d, other.d); }

private:
    friend void swap(Pen& lhs, Pen& rhs) noexcept { lhs.swap(rhs); }
    ~~~
```

.cpp

```
struct Pen::Private {
    std::atomic<int> ref;
    ~~~
};
Pen::Pen(const Pen& other) : d(other.d) {
    assert(d); // no copying from partially-formed
    ++d->ref;
}
Pen::~Pen() {
    if (d && !--d->ref)
        delete d;
}
```

A Case Study: Implicitly-Shared Pen II: domain members

.h

```
~~~
    void detach();

public:
    static Pen solid(Color c, int thickness);

    Color color() const;
    void setColor(Color c);
};
```

.cpp

```
struct Pen::Private {
    std::atomic<int> ref;
    Color color;
    int thickness;
    ~~~
};
void Pen::detach() {
    // ???
    if (d->ref != 1)
        d = new Private{*d}; // modulo std::atomic
}
Pen Pen::solid(Color c, int thickness) {
    Pen result;
    result.d = new Private{1, c, thickness};
    return result;
}
Color Pen::color() const {
    assert(d); // not allowed on partially-formed
    return d->color;
}
void Pen::setColor(Color c) {
    detach();
    d->color = c;
}
```

A Case Study: Implicitly-Shared Pen II: domain members

.h

```
~~~
    void detach();

public:
    static Pen solid(Color c, int thickness);

    Color color() const;
    void setColor(Color c);
};
```

.cpp

```
struct Pen::Private {
    std::atomic<int> ref;
    Color color;
    int thickness;
    ~~~
};
void Pen::detach() {
    assert(d); // no detaching from partially-formed
    if (d->ref != 1)
        d = new Private{*d}; // modulo std::atomic
}
Pen Pen::solid(Color c, int thickness) {
    Pen result;
    result.d = new Private{1, c, thickness};
    return result;
}
Color Pen::color() const {
    assert(d); // not allowed on partially-formed
    return d->color;
}
void Pen::setColor(Color c) {
    detach();
    d->color = c;
}
```

Bonus Slides

- A Case Study: Pen
- **Weaker Models**
- Weak Exception Guarantees

Overview of Move Semantics Models

- std::move() + valid-but-unspecified
- std::move() + partially-formed
- std::pilfer() + pilfered (destroy-only; P0308)
- destructive move

Guideline 11

Move semantics violate the Zero-Overhead-Rule (D&E, P0559).
Partially-Formed States are the natural states
in the move semantics model we have.

Bonus Slides

- A Case Study: Pen
- Weaker Models
- **Weak Exception Guarantees**

Abrahams' Exception Guarantees

- basic
  - no resource leaks
  - all invariants maintained
    - "valid, but unspecified"
- strong
  - transactional semantics
- nothrow

Abrahams' Exception Guarantees Extended

- weak
  - no resource leaks
  - objects are Partially-Formed
- basic
  - weak + all invariants maintained
- strong
- nothrow

`std::variant`

"Weak" Guarantee is probably not needed

- diff is only in the docs!

But then came `std::variant::valueless_by_exception()`

- Wouldn't be needed if we had the weak guarantee instead

Thank you for your attention (now for real)!

Questions?