# The C++ rvalue lifetime disaster

*Arno Schoedl*

# The C++ Rvalue Lifetime Disaster

by Arno Schoedl

aschoedl@think-cell.com

# Rvalue References

- STL advocates value semantics
- lead to frequent copying in C++03
- rvalue references invented to avoid copying
  - replaced by more efficent moving

# Rvalue References

- STL advocates value semantics
- lead to frequent copying in C++03
- rvalue references invented to avoid copying
  - replaced by more efficent moving

```
std::vector< std::vector<int> > vecvec;
std::vector<int> vec={1,2,3};
vecvec.emplace_back( std::move(vec) ); // rvalue reference avoids copy
```

# Rvalue References

- STL advocates value semantics
- lead to frequent copying in C++03
- rvalue references invented to avoid copying
  - replaced by more efficent moving

```
std::vector< std::vector<int> > vecvec;
std::vector<int> vec={1,2,3};
vecvec.emplace_back( std::move(vec) ); // rvalue reference avoids copy
```

- Increasingly used to manage lifetime
  - C++11 `std::cref`
  - C++20 Ranges

# Rvalue References

- STL advocates value semantics
- lead to frequent copying in C++03
- rvalue references invented to avoid copying
  - replaced by more efficent moving

```
std::vector< std::vector<int> > vecvec;
std::vector<int> vec={1,2,3};
vecvec.emplace_back( std::move(vec) ); // rvalue reference avoids copy
```

- Increasingly used to manage lifetime
  - C++11 `std::cref`
  - C++20 Ranges

```
auto rng=std::vector<int>{1,2,3} | std::views::filter([](int i){ return 0==i%2
; }); // DOES NOT COMPILE
```

- `rng` would contain dangling reference to `std::vector<int>`
- So `std::views::filter` does not compile for rvalues

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A const a=...;
  ...
  return std::move(a);
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A const a=...;
  ...
  return std::move(a);
};
```

- What happens?
  - Copy - cannot move out of `const`

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A const a=...;
  ...
  return std::move(a);
};
```

- What happens?
  - Copy - cannot move out of `const`

```
A foo() {
  A a=...;
  ...
  return std::move(a);
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A const a=...;
  ...
  return std::move(a);
};
```

- What happens?
  - Copy - cannot move out of `const`

```
A foo() {
  A a=...;
  ...
  return std::move(a);
};
```

- What happens?
  - Move
  - Best we can do?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A a=...;
  ...
  return a;
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A a=...;
  ...
  return a;
};
```

- What happens?
  - NRVO (Named Return Value Optimization) - copy/move elided
  - `std::move` can make things worse

# Rvalue References for Moving - Pitfalls

```
A foo() {
  A a=...;
  ...
  return a;
};
```

- What happens?
  - NRVO (Named Return Value Optimization) - copy/move elided
  - `std::move` can make things worse

```
A foo() {
  A const a=...;
  ...
  return a;
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  A a=...;
  ...
  return a;
};
```

- What happens?
  - NRVO (Named Return Value Optimization) - copy/move elided
  - `std::move` can make things worse

```
A foo() {
  A const a=...;
  ...
  return a;
};
```

- What happens?
  - Still NRVO (Named Return Value Optimization) - copy/move elided

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  if(... condition ...) {
    A const a=...;
    ...
    return a;
  } else {
    A const a=...;
    ...
    return a;
  }
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  if(... condition ...) {
    A const a=...;
    ...
    return a;
  } else {
    A const a=...;
    ...
    return a;
  }
};
```

- What happens?
  - No NRVO, returned object is not always same one
  - Copy because of `const` :-(

# Rvalue References for Moving - Pitfalls

think-cell

```
A foo() {
  if(... condition ...) {
    A a =...;
    ...
    return a;
  } else {
    A a=...;
    ...
    return a;
  }
};
```

- What happens?
  - Move

# Rvalue References for Moving - Pitfalls

think-cell

```
struct B {
  A m_a;
};

A foo() {
  B b=...;
  ...
  return b.m_a;
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

think-cell

```
struct B {
  A m_a;
};

A foo() {
  B b=...;
  ...
  return b.m_a;
};
```

- What happens?
  - Copy
  - Members do not automatically become rvalues

# Rvalue References for Moving - Pitfalls

```
struct B {
  A m_a;
};

A foo() {
  B b=...;
  ...
  return std::move(b).m_a;
};
```

- What happens?

# Rvalue References for Moving - Pitfalls

```
struct B {
  A m_a;
};

A foo() {
  B b=...;
  ...
  return std::move(b).m_a;
};
```

- What happens?
  - Move
  - Member access of rvalue is rvalue

# Rvalue References for Moving - Pitfalls

think-cell

```
struct B {
  A m_a;
};

A foo() {
  B b=...;
  ...
  return std::move(b).m_a;
};
```

- What happens?
  - Move
  - Member access of rvalue is rvalue
- Recommendations
  - Make return variables non-`const`
  - Use Clang's `-Wmove`

# Temporary Lifetime Extension

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const& { return m_a; }
};

B b;
auto const& a=b.getA();
```

# Temporary Lifetime Extension

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const& { return m_a; }
};

struct C {
  A getA() const&;
};

B b;
C c;
auto const& a=< b or c >.getA();
```

# Temporary Lifetime Extension

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const& { return m_a; }
};

struct C {
  A getA() const&;
};

B b;
C c;
auto const& a=< b or c >.getA();
```

- `auto const& a=c.getA();` works thanks to *temporary lifetime extension*
- Idea: always write `auto const&`, the right thing happens

# Temporary Lifetime Extension vs. Rvalues

```
bool operator<(A const&, A const&);

struct C {
  A getA() const&;
} c1, c2;
auto const& a=std::min( c1.getA(), c2.getA() );
```

# Temporary Lifetime Extension vs. Rvalues

```
bool operator<(A const&, A const&);

struct C {
  A getA() const&;
} c1, c2;
auto const& a=std::min( c1.getA(), c2.getA() );
```

```
namespace std {
  template<typename T>
  T const& min( T const& lhs, T const& rhs ) {
    return rhs<lhs ? rhs : lhs;
  }
}
```

- `std::min` forgets about rvalue-ness
- `a` dangles

# Temporary Lifetime Extension vs. Rvalues

```
bool operator<(A const&, A const&);

struct C {
  A getA() const&;
} c1, c2;
auto const& a=our::min( c1.getA(), c2.getA() );
```

```
namespace our {
  template<typename Lhs, typename Rhs>
  decltype(auto) min( Lhs&& lhs, Rhs&& rhs ) {
    return rhs<lhs ? std::forward<Rhs>(rhs) : std::forward<Lhs>(lhs);
  }
}
```

- `our::min` correctly returns `A&&`

# Temporary Lifetime Extension vs. Rvalues

```
bool operator<(A const&, A const&);

struct C {
  A getA() const&;
} c1, c2;
auto const& a=our::min( c1.getA(), c2.getA() );
```

```
namespace our {
  template<typename Lhs, typename Rhs>
  decltype(auto) min( Lhs&& lhs, Rhs&& rhs ) {
    return rhs<lhs ? std::forward<Rhs>(rhs) : std::forward<Lhs>(lhs);
  }
}
```

- `our::min` correctly returns `A&&`
- `a` still dangles
- *temporary lifetime extension does not keep rvalue references alive!*
  - would only be possible by creating a copy

# Temporary Lifetime Extension vs. decltype(auto)

think-cell

```
A some_A();
- or -
A const& some_A();
```

- forwarding return:

```
decltype(auto) foo() {
  return some_A();
}
```

# Temporary Lifetime Extension vs. decltype(auto)

```
A some_A();
- or -
A const& some_A();
```

- forwarding return:

```
decltype(auto) foo() {
  return some_A();
}
```

- forwarding return with code in between:

```
??? foo() {
  ??? a = some_A();
  ... do something ...
  return a;
}
```

# Temporary Lifetime Extension vs. decltype(auto)

```
decltype(auto) foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

# Temporary Lifetime Extension vs. decltype(auto)

```
decltype(auto) foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- creates dangling reference if `some_A()` returns value

# Temporary Lifetime Extension vs. decltype(auto)

```
decltype(auto) foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- creates dangling reference if `some_A()` returns value

```
auto foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

# Temporary Lifetime Extension vs. decltype(auto)

```
decltype(auto) foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- creates dangling reference if `some_A()` returns value

```
auto foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- always copies

# Temporary Lifetime Extension vs. decltype(auto)

```
decltype(auto) foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- creates dangling reference if `some_A()` returns value

```
auto foo() {
  auto const& a = some_A();
  ... do something ...
  return a;
}
```

- always copies
- Problem: temporary lifetime extension lies about its type
  - if `some_A()` returns value, `a` is really value, not reference

# auto_cref

- Deprecate temporary lifetime extension
- Automatically declare variable
  - `auto` if constructed from value or rvalue reference, and
  - `auto const&` if constructed from lvalue reference

# auto_cref

- Deprecate temporary lifetime extension
- Automatically declare variable
  - `auto` if constructed from value or rvalue reference, and
  - `auto const&` if constructed from lvalue reference

```
template<typename T>
struct decay_rvalues;

template<typename T>
struct decay_rvalues<T&> {
  using type=T&;
};

template<typename T>
struct decay_rvalues<T&&> {
  using type=std::decay_t<T>;
};

#define auto_cref( var, ... ) \
  typename decay_rvalues<decltype((__VA_ARGS__))&&>::type var = ( __VA_ARGS__
 );
```

# auto_cref

think-cell

```
decltype(auto) foo() {
  auto_cref( a, some_A() );
  ... do something with a ...
  return a;
}
```

# auto_cref

```
decltype(auto) foo() {
  auto_cref( a, some_A() );
  ... do something with a ...
  return a; // no parentheses here!
}
```

# auto_cref

```
decltype(auto) foo() {
  auto_cref( a, some_A() );
  ... do something with a ...
  return a; // no parentheses here!
}
```

- Make it your default `auto` !
  - does not work yet if expression contains lambda, fixed in C++20

# auto_cref

```
decltype(auto) foo() {
  auto_cref( a, some_A() );
  ... do something with a ...
  return a; // no parentheses here!
}
```

- Make it your default `auto` !
  - does not work yet if expression contains lambda, fixed in C++20
- Choice: `auto_cref` value `const`?

```
template<typename T>
struct decay_rvalues<T&&> {
using type=std::decay_t<T> const;
};
```

- Then `auto_cref_return` for NRVO/move optimization

# auto_cref

```
bool operator<(A const&, A const&);

struct C {
  A getA() const&;
} c1, c2;

auto_cref( a, our::min( c1.getA(), c2.getA() ) );
```

```
namespace our {
  template<typename Lhs, typename Rhs>
  decltype(auto) min( Lhs&& lhs, Rhs&& rhs ) {
    return rhs<lhs ? std::forward<Rhs>(rhs) : std::forward<Lhs>(lhs);
  }
}
```

- `our::min` correctly returns rvalue reference
- `auto_cref` correctly turns it into value

# C++ Rvalue Amnesia

```
struct A;

struct B {
  A m_a;
};

auto_cref( a, B().m_a );
```

# C++ Rvalue Amnesia

```
struct A;

struct B {
  A m_a;
};

auto_cref( a, B().m_a );
```

- Works
  - `decltype((B().m_a))` is `A&&`
  - `a` is value

# C++ Rvalue Amnesia

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const {
    return m_a;
  }
};

auto_cref( a, B().getA() );
```

# C++ Rvalue Amnesia

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const {
    return m_a;
  }
};

auto_cref( a, B().getA() );
```

- Does not work
  - `decltype(B().getA())` is `A const&`
  - `a` is `const&`, dangles

# C++ Rvalue Amnesia

think-cell

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const& {
    return m_a;
  }
};

auto_cref( a, B().getA() );
```

- Does not work
  - `decltype(B().getA())` is `A const&`
  - `a` is `const&`, dangles

# C++ Rvalue Amnesia

think-cell

```
struct A;

struct B {
private:
  A m_a;
public:
  A const& getA() const& {
    return m_a;
  }
};

auto_cref( a, B().getA() );
```

- Does not work
  - `decltype(B().getA())` is `A const&`
  - `a` is `const&`, dangles
- Fundamental problem: `const&` binds anything, including rvalues
- Affects any `const&` accessor

# Conditional Operator Afraid Of Rvalue Amnesia

```
struct A;
A const& L();
A const&& R();
```

- What is `decltype( false ? L() : L() )`?
  - `A const&`
- What is `decltype( false ? R() : R() )`?
  - `A const&&`

# Conditional Operator Afraid Of Rvalue Amnesia

```
struct A;
A const& L();
A const&& R();
```

- What is `decltype( false ? L() : L() )`?
  - `A const&`
- What is `decltype( false ? R() : R() )`?
  - `A const&&`
- What is `decltype( false ? R() : L() )`?

# Conditional Operator Afraid Of Rvalue Amnesia

```
struct A;
A const& L();
A const&& R();
```

- What is `decltype( false ? L() : L() )`?
  - `A const&`
- What is `decltype( false ? R() : R() )`?
  - `A const&&`
- What is `decltype( false ? R() : L() )`?
  - `A const`
  - C++ forces a copy

# C++20 common_reference Not Afraid

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges

# C++20 common_reference Not Afraid

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`

# C++20 common_reference Not Afraid

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is

# C++20 common_reference Not Afraid

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is
  - `A const&` !

# C++20 common_reference Not Afraid

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is
  - `A const&` !
- `std::common_reference_t< A const, A const& >` is

# C++20 common_reference Not Afraid

think-cell

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is
  - `A const&` !
- `std::common_reference_t< A const, A const& >` is
  - `A` !

# C++20 common_reference Not Afraid

think-cell

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is
  - `A const&` !
- `std::common_reference_t< A const, A const& >` is
  - `A` !
- `std::common_reference` embraces rvalue amnesia

# C++20 common_reference Not Afraid

think-cell

- C++20 has new trait `common_reference_t`
  - invented for C++20 Ranges
- `std::common_reference_t< A const&, A const& >` is
  - `A const&`
- `std::common_reference_t< A const&&, A const&& >` is
  - `A const&&`
- `std::common_reference_t< A const&&, A const& >` is
  - `A const&` !
- `std::common_reference_t< A const, A const& >` is
  - `A` !
- `std::common_reference` embraces rvalue amnesia

WHAT IS CORRECT?

# Promises of References

think-cell

| Mutablity \ Lifetime | **short** | **long** |
|---|---|---|
| immutable | **const**&& | **const**& |
| **mutable** | && | & |

# Promises of References

| Mutablity \ Lifetime | **short** | **long** |
| --- | --- | --- |
| immutable | **const**&& | **const**& |
| **mutable** | && [can scavenge] | & |

# Promises of References

```
              Lifetime    short               long
Mutablity

immutable                 const&& ------>  const&
                                       ->
                             ^       /        ^
                             |    --          |
                               /
mutable                    &&                 &
                       [can scavenge]
```

- Current C++ reference binding strengthens lifetime promise

# Promises of References

```
           Lifetime    short           long
Mutablity

immutable              const&& <-----  const&
                               <-
                          ^       \      ^
                          |         --   |
                                       \
mutable                  &&              &
                   [can scavenge]
```

- Better: Allow binding only if promises get weaker
  - less lifetime
  - less mutability
  - less "scavenge-ability"
- only lvalues should bind to `const&`
- anything may bind to `const&&`

# UUuuuuuuuh

- This is so sad.
- It is very sad.
- We dug ourselves a hole.
- And fell into it.
- UUuuuuh.

# Any Chance to Fix C++?

- Warning: These are Ideas! Has not been Implemented!

# Any Chance to Fix C++?

- Warning: These are Ideas! Has not been Implemented!
- Existing code must continue to work
- Existing libraries must work with new code
  - gradual introduction of new binding rules within one codebase

# Any Chance to Fix C++?

think-cell

- Warning: These are Ideas! Has not been Implemented!
- Existing code must continue to work
- Existing libraries must work with new code
    - gradual introduction of new binding rules within one codebase
- Any reference uses either new or old rules
    - Reference binding only at beginning of reference lifetime
    - Type of resulting reference unchanged

# Any Chance to Fix C++?

- Warning: These are Ideas! Has not been Implemented!
- Existing code must continue to work
- Existing libraries must work with new code
  - gradual introduction of new binding rules within one codebase
- Any reference uses either new or old rules
  - Reference binding only at beginning of reference lifetime
  - Type of resulting reference unchanged
- All declarations inside `#new-reference-binding on/off` bind along new rules

```
auto const& a = ... // old rules apply
#new-reference-binding on
auto const& b = ... // new rules apply
#new-reference-binding off
auto const& c = ... // old rules apply
```

# Reference Declarations (1)

think-cell

- local/global variable initialization

```
auto const& a = ...
```

- structured binding

```
auto const& [a,b] = ...
```

- function/lambda parameter lists

```
void foo(A const& a);
```

# Reference Declarations (2)

- members (initialized in PODs)

```
struct B {
  A const& m_a;
} b = { a };
```

- members (initialized in constructors)

```
struct B {
  A const& m_a;
  B(A const& a) : m_a(a) {}
};
```

- lambda captures

```
[&a = b]() { .... };
```

# How to opt in to new behavior?

- All declarations inside `#new-reference-binding on/off` bind along new rules

```
void A(int const& a);

#new-reference-binding on
void B(int const& a);
void C(int const&& a);

#new-reference-binding off
void B(int const& a) { // error: declared with different binding behavior
    ...
}

A(5); // compiles
B(5); // error: cannot bind rvalue to lvalue
C(5); // compiles

int a=1;
C(a); // compiles
```

# Impact on Standard Library

- Feature-test macro if `#new-reference-binding` is enabled
- Functions can be implemented equivalently
  - typically replace `const&` parameters with `const&&`
- `<type_traits>`
  - `std::common_reference`
  - others not affected

# Until then... Mitigations (1)

- temporary lifetime extension
  - replace by `auto_cref`

# Until then... Mitigations (1)

- temporary lifetime extension
  - replace by `auto_cref`
- member accessors
  - delete rvalue accessors
  - macro?

```
struct B {
private:
  A m_a;
public:
  A const& getA() const& {
    return m_a;
  }
  A const& getA() const&& = delete;
};
```

# Until then... Mitigations (2)

- `common_reference`

```
namespace our {
  template<typename... Ts>
  struct common_reference {
    using oldtype=std::common_reference_t<Ts...>;
    using type=std::conditional_t<
      std::is_lvalue_reference<oldtype>::value &&
        std::disjunction<std::is_rvalue_reference<Ts> ...>::value,
      std::remove_reference_t<oldtype>&&,
      oldtype
    >;
  };
}
```

# Until then... Mitigations (3)

- `decltype( false ? R() : L() )`?
  - `A const`
  - C++ forces a copy

# Until then... Mitigations (3)

think-cell

- `decltype( false ? R() : L() )`?
  - `A const`
  - C++ forces a copy
- `our::common_reference` allows fearless conditional (ternary) operator

```
#define CONDITIONAL(b, l, r) ( \
    b \
    ? static_cast< typename our::common_reference<decltype((l)),decltype((r))
>::type >(l) \
    : static_cast< typename our::common_reference<decltype((l)),decltype((r))
>::type >(r) \
)
```

# Until then... Mitigations (3)

think-cell

- `decltype( false ? R() : L() )`?
  - `A const`
  - C++ forces a copy
- `our::common_reference` allows fearless conditional (ternary) operator

```
#define CONDITIONAL(b, l, r) ( \
    b \
    ? static_cast< typename our::common_reference<decltype((l)),decltype((r))
>::type >(l) \
    : static_cast< typename our::common_reference<decltype((l)),decltype((r))
>::type >(r) \
)
```

- `decltype( CONDITIONAL( false, R(), L() ) )`?
  - `A const&&`
  - no immediate copy

# Summary

- `const&` should never have bound to rvalues
- Fixing C++ may be possible, but must demonstrate it
  - Clang implementation
  - large code base to try it on
- Until then, consider mitigations

THANK YOU!