When the mix of modern C++ features surprise you

The journey covered

- non compiling unique_ptr examples
- copy elision mechanics and rules
- RVO with move semantics
- changes and defects in C++11, C++14, C++17, C++20
- new proposals enhancing nr of contexts for copy elision and move object creation on fun return

Journey memoirs

- you can't blindly rely on what your compiler tells you
- upgrade compilers in your projects
- upgrade C++ standard
- less time on investigating bugs
- increased performance
- cleaner codebase

References

[00] David Stone - P0527R1 Implicitly move from rvalue references in return statements
[01] Arthur O'Dwyer, David Stone - P1155R2 More implicit moves
[02] Antony Polukhin - P0889R1 Ultimate copy elision
[03] Arthur O'Dwyer - RVO harder than it looks - CppCon 2018
[04] John Kalb - Copy elision - CppCon 2018
[05] Jonas Devlieghere - Guaranteed Copy Elision
[06] Simon Brand - Guaranteed Copy Elision Does Not Elide Copies
[07] David Rodriguez - Value semantics: Copy elision
[08] David Rodriguez - Value semantics: NRVO
[09] Zhao Wu IBM - RVO V.S. std::move
[10] Scott Meyers - Effective Modern C++
[11] A proposal to add move semantics support to the C++ Language - 2002
[12] C++11 RVO defect report
[13] cppreference - return statement
[14] cppreference - copy elision
[15] cppreference - value category

Thank you

Lukasz Bondyra

Senior Engineer @ Motorola Solutions

https://github.com/bondyr/rvo_ppt

Lkbondyra@gmail.com

A. Teaching practice

- code with numerous functions since we've been taught for decades to delegate to funs and classes
- compilers inline more aggressievly and inline a lot

B. Copy elision rules

- current rules for copy elision mostly assume that a function from source code remains a function in a binary
- inlining, aliasing reasoning, and link time optimization open a new world to possible optimizations since fun params and its body is seen together
- current rules are suboptimal for modern compilers: they prevent optimizations.

C. std::move/rvalue is not a panacea

move operation is not a solution since

std::array

std::basic_string
std::function
std::variant
std::optional
and others

may store a lot of data on stack

Summary

- year 2002 - the oldest proposal on move semantics found

A Proposal to Add Move Semantics Support to the C++ Language

- move semantics present for almost 2 decades now
- there're still areas to optimize

- Slicing is pessimized

```
Base thirteen() {
    Derived result;
    return result;
}
```

Different object's type -
no copy elision, and no implicit move (the object is copied)

Note:

C++17 standard wording	- copy
Clang, MSVC	- copy
GCC 8.1+, ICC 18.0.0+	- implicit move

Current iso cpp wording has issues

"We propose to remove all four of these unnecessary limitations."

- implemented in clang in 02.2018
- -Wreturn-std-move (enabled as part of -Wmove, -Wmost, and -Wall)
- overload resolution done twice - standard rules and the ones proposed here
- if they differ, then diagnostic warning is emitted suggesting adding explicit std::move since copy otherwise

Feedback from production

Both papers merged as P1825 and included in C++20

http://wg21.link/p1825

Plenty of true positives

Proven with clang flag enabled on Chromium and LibreOffice projects

p0527

Lack of false positives

p1155

Just one complaint from Mozzilla saying that performance benefit was too little compared to ugliness of explicit std::move. Actually it's not even mistake.

P0889
Ultimate copy elision

Antony Polukhin

http://wg21.link/p0889

- Throwing is pessimized

```
void seven(Widget w) {
    throw w;
}
```

Throw function parameter -
no copy elision, and no implicit move (the object is copied)

Note:

C++17 standard wording	- copy
GCC	- copy
Clang 4.0.1+, MSVC 2015+, ICC 16.0.3+	- implicit move

- Similar suggestions regarding rvalue reference function parameters used in throw expression and exception declaration.
- Wording change to include rvalue reference:
"an object with automatic storage duration" -> "a movable entity"

Proposal included in C++20 standard.

- Non-constructor conversion is pessimized

```
struct To {
    operator Widget() const &;
    operator Widget() &&;
};

Widget nine() {
    To t;
    return t;
}
```

Conversion operator instead of ctor -
no copy elision, and no implicit move (the object is copied)

- By-value sinks are pessimized

```
struct Fowl {
    Fowl(Widget);
};

Fowl eleven() {
    Widget w;

    return w;
}
```

Value instead of rvalue ref -
no copy elision, and no implicit move (the Widget object is copied)

Note:

C++17 standard wording	- copy
Clang, ICC, MSVC	- copy
GCC 5.1+	- implicit move

Surprising results

```
// move ctor
string f(std::string x) { return x; }

// copy ctor - quite surprising...
string g(string && x) { return x; }
```

P1155
More implicit moves

Arthur O'Dwyer
David Stone

http://wg21.link/p1155

Paper starts with:

"Programmers expect return x; to trigger copy elision; or, at worst, to implicitly move from x instead of copying. Occasionally, C++ violates their expectations and performs an expensive copy anyway. [...]"

```
unique_ptr fun(unique_ptr && ptr) {
    return ptr;
}
```

error: use of deleted function
'unique_ptr::unique_ptr(const unique_ptr&)'

could be fixed by:

```
return move(ptr);
```

consider generic case

```
auto fun(args...) {
    decltype(auto) result = some_other_fun();
    return forward<decltype(result)>(result);
}
```

with std::forward, RVO inhibited when result is a value and not a reference

without std::forward, doesn't compile when result is rvalue ref and is move only

Current/future RVO optimization proposals

P0527
Implicitly move from rvalue references in return statements

David Stone

http://wg21.link/p0527

When the mix of modern C++ features surprise you

Copy elision recall

Move semantics recall

Back to story

Let's drop unique_ptr for few secs

Another compilation error

auto

unique_ptr

move

RVO

c++11

gcc 4.9.2

custom make_unique()

www.godbolt.org

Summary:

unique_ptr to class hierarchy doesn't compile on gcc 4.9.4

it compiles on gcc 5.1+

object slicing performs copy on gcc 8.0-

object slicing performs move on gcc 8.1+

object slicing performs copy on clang 8.0.0

Pretty messed up

Copy elision

Branching

copy elision combined

What's the magic behind copy elision

How many params fun() takes?

0 params

1 param

- the caller allocates space on stack for the return value
- and passes this address to the callee
- the callee constructs directly in that space

Copy elision - extracted

- automatic storage
- the same type
- results "as if" copies were made

C++ standard wording

RVO, NRVO

Init objects with temporary

No copy elision

Copy elision

RVO in C++17

"Guaranteed copy elision through simplified value categories"

Actually...

When copy elision not eligible...

copy or move

Return statement

Back to object slicing copy/move problem

Why gcc 8.1+ uses move?

Move or copy?

So why object slicing compiled on old compiler?

C++11* criteria for RVO with move

DR1579

DR1579 resolution

auto

unique_ptr

move

RVO

Copy elision recall

```
auto fun1()
{
    return std::vector<int>(100);
    // RVO (Return Value Optimization)
}

auto fun2()
{
    std::vector<int> vec(100);
    return vec;
    // Named RVO
}

auto vec = fun1();   // only one ctor call - copy elided
auto vec = fun2();   // only one ctor call - copy elided
```

Move semantics recall

```
std::vector<int> vec(100);

auto vec_copy = vec;
auto vec_move = std::move(vec);



vector::vector(const T&) - copy ctor overload resolution for lvalues
vector::vector(T&&)      - move ctor overload resolution for rvalues


std::move    - only casts to rvalue reference

std::forward - conditional move
```

Back to story

```
class Combiner { ... };

unique_ptr<Combiner> create()
{
    auto combiner = make_unique<Combiner>();
    combiner->init();
    ...
    return move(combiner);
}
```

Things to Remember

- Apply `std::move` to rvalue references and `std::forward` to universal references the last time each is used.
- Do the same thing for rvalue references and universal references being returned from functions that return by value.
- Never apply `std::move` or `std::forward` to local objects if they would otherwise be eligible for the return value optimization.

Back to story

```
class Combiner { ... };

unique_ptr<Combiner> create()
{
    auto combiner = make_unique<Combiner>();
    combiner->init();
    ...
    return move(combiner);
}
```

```
class MixerInterface{...};
class Mixer : public MixerInterface {...};

unique_ptr<MixerInterface> create()
{
    auto mixer = make_unique<Mixer>();
    mixer->init();
    ...
    return move(mixer);
}
```

```
class MixerInterface{...};
class Mixer : public MixerInterface {...};

unique_ptr<MixerInterface> create()
{
    auto mixer = make_unique<Mixer>();
    mixer->init();
    ...
    return mixer;
}
```

```
error: cannot bind 'unique_ptr<Mixer>' lvalue
                to 'unique_ptr<Mixer>&&'
```

```
note: initializing argument 1 of
'unique_ptr<_Tp, _Dp>::unique_ptr(unique_ptr<_Up, _Ep>&&)
[with _Up = Mixer;          _Ep = default_delete;
      _Tp = MixerInterface; _Dp = default_delete]'



unique_ptr<MixerInterface>::unique_ptr(unique_ptr<Mixer>&& u);
```

```
class MixerInterface{...};
class Mixer : public MixerInterface {...};

unique_ptr<MixerInterface> create()
{
    auto mixer = make_unique<Mixer>();
    mixer->init();
    ...
    return mixer;
}
```

```
error: cannot bind 'unique_ptr<Mixer>' lvalue
                to 'unique_ptr<Mixer>&&'
```

So debugging starts...

```
A)
unique_ptr<MixerInterface> create()
{
    unique_ptr<Mixer> mixer = make_unique<Mixer>();
    return mixer;
    // compiler error, as before
}


B)
unique_ptr<MixerInterface> create()
{
    unique_ptr<MixerInterface> mixer = make_unique<Mixer>();
    return mixer;
    // OK - copy elision (NRVO)
}
```

C)

```
unique_ptr<MixerInterface> create()
{
    return make_unique<MixerInterface>();
    // OK - copy elision (RVO)
}
```

D)

```
unique_ptr<MixerInterface> create()
{
    return make_unique<Mixer>();
    // OK!
    // And move ctor
}
```

Weird...

```
A)
unique_ptr<MixerInterface> create()
{
    unique_ptr<Mixer> mixer = make_unique<Mixer>();
    return mixer;
    // compiler error, as before
}


D)
unique_ptr<MixerInterface> create()
{
    return make_unique<Mixer>();
    // OK!
    // And move ctor
}
```

Another compilation error

```
E)
unique_ptr<const MixerInterface> create()
{
    auto mixer = make_unique<MixerInterface>();
    mixer->init();
    ...
    return mixer;
}


error: cannot bind 'unique_ptr<MixerInterface>' lvalue
                to 'unique_ptr<MixerInterface>&&'


unique_ptr<const MixerInterface>::
unique_ptr(unique_ptr<MixerInterface>&& u);
```

Let's drop unique_ptr for few secs

```
F)
MixerInterface create()
{
    return Mixer();
    // OK - move ctor
}

G)
MixerInterface create()
{
    Mixer x;
    return x;
    // OK - copy ctor

}
```

c++11

gcc 4.9.2

custom make_unique()

www.godbolt.org

non-compiling examples on gcc 5.1+

```
A)
unique_ptr<MixerInterface> create()
{
    auto mixer = make_unique<Mixer>();
    return mixer;
    // OK - move ctor
}



E)
unique_ptr<const MixerInterface> create()
{
    auto mixer = make_unique<MixerInterface>();
    return mixer;
    // OK - move ctor


}
```

object slicing on gcc 8.1+

```
G)
MixerInterface create()
{
    Mixer x;
    return x;
    // OK - move ctor now!
}
```

```
49 create():
50         push    rbp
51         mov     rbp, rsp
52         sub     rsp, 32
53         mov     QWORD PTR [rbp-24], rdi
54         lea     rax, [rbp-1]
55         mov     rdi, rax
56         call    Mixer::Mixer() [complete object constructor]
57         lea     rdx, [rbp-1]
58         mov     rax, QWORD PTR [rbp-24]
59         mov     rsi, rdx
60         mov     rdi, rax
61         call    MixerInterface::MixerInterface(MixerInterface&&)
62         mov     rax, QWORD PTR [rbp-24]
63         leave
64         ret
```

object slicing on clang 8.0.0

```
G)
MixerInterface create()
{
    Mixer x;
    return x;
    // OK - still copy ctor
}
```

```
16 create():                               # @create()
17         push    rbp
18         mov     rbp, rsp
19         sub     rsp, 32
20         mov     rax, rdi
21         lea     rcx, [rbp - 8]
22         mov     qword ptr [rbp - 16], rdi # 8-byte Spill
23         mov     rdi, rcx
24         mov     qword ptr [rbp - 24], rax # 8-byte Spill
25         call    Mixer::Mixer() [base object constructor]
26         lea     rax, [rbp - 8]
27         mov     rdi, qword ptr [rbp - 16] # 8-byte Reload
28         mov     rsi, rax
29         call    MixerInterface::MixerInterface(MixerInterface const&)
30         mov     rax, qword ptr [rbp - 24] # 8-byte Reload
31         add     rsp, 32
32         pop     rbp
33         ret
```

Summary:

unique_ptr to class hierarchy doesn't compile on gcc 4.9.4-

it compiles on gcc 5.1+

object slicing performs copy on gcc 8.0-

object slicing performs move on gcc 8.1+

object slicing performs copy on clang 8.0.0

Pretty messed up

Copy elision

- mechanism that is permitted under certain conditions

- allowed, not required

- omit copy/move ctors

- even if ctor/dtor have side efects

- objects constructed directly in the storage they'd be copied to

- copy/move ctors need to be present anyway

- not in debug mode (-g flag)

- portability vs ctor/dtor side effects

C++ standard wording

31 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.[123] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

(31.1) — in a **`return`** statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value

(31.2) — in a *throw-expression*, when the operand is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing *try-block* (if there is one), the copy/move operation from the operand to the exception object (15.1) can be omitted by constructing the automatic object directly into the exception object

(31.3) — when a temporary class object that has not been bound to a reference (12.2) would be copied/moved to a class object with the same cv-unqualified type, the copy/move operation can be omitted by constructing the temporary object directly into the target of the omitted copy/move

(31.4) — when the *exception-declaration* of an exception handler (Clause 15) declares an object of the same type (except for cv-qualification) as the exception object (15.1), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration*. [*Note:* There cannot be a move from the exception object because it is always an lvalue. — *end note*]

Copy elision - extracted

- automatic storage

- the same type

- results "as if" copies were made

RVO, NRVO

```
Mixer fun1() {
     return Mixer{};
     // RVO
}

Mixer fun2() {
     Mixer x;
     return x;
     // NRVO
}
```

init objects with temporary

```
void fun3(Mixer x) {...}

int main() {
    // create arg with temp object
    fun3(Mixer{});
}



// create var with temp object
Mixer x1 = Mixer{};
```

copy elision combined

```
// RVO + create var with temp object
Mixer x2 = fun1();


// RVO + create arg with temp object
fun3(fun1());
```

Branching

```
// no RVO
Mixer create1(int q) {
    Mixer x1, x2;
    if(q > 4)
        return x1;
    else
        return x2;
}
```

```
// optimized for RVO
Mixer create2(int q) {
    if(q > 4) {
        Mixer x1;
        return x1;
    }
    else {
        Mixer x2;
        return x2;
    }
}
```

```
bool pred(const Mixer& x1, const Mixer& x2)
{
    return x1 > x2;
}


// no RVO possible
Mixer create3() {
    Mixer x1, x2;
    if(pred(x1,x2))
        return x1;
    else
        return x2;
}
```

Never depend on side effects in move/copy ctor, as small change in code can allow or inhibit RVO.

What's the magic behind copy elision

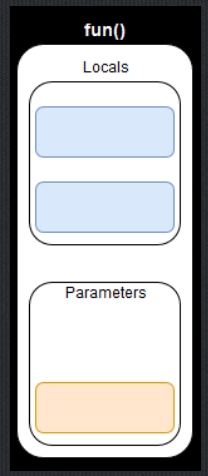
```
Mixer fun() {
    int z;
    Mixer x;
    return x;
}

Mixer y{fun()};
```

How many params fun() takes?

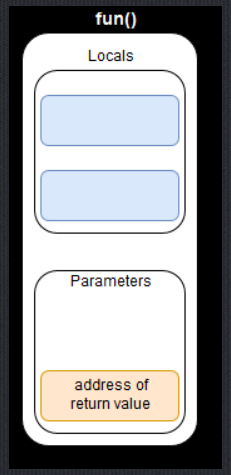
0 params

?

1 param



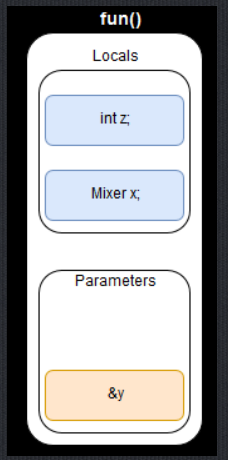
1 param



- the caller allocates space on stack for the return value

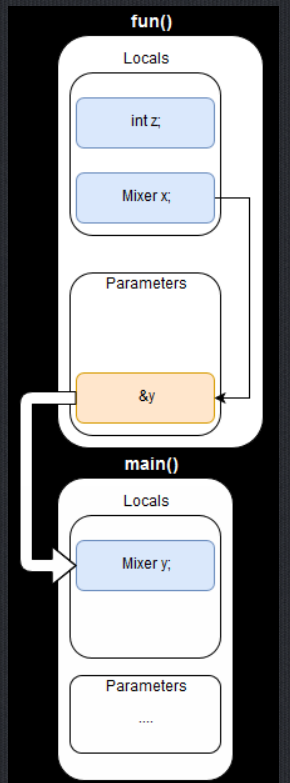
- and passes this address to the callee
- the callee constructs directly in that space

```
Mixer fun() {
    int z;
    Mixer x;
    return x;
}

Mixer y{fun()};
```



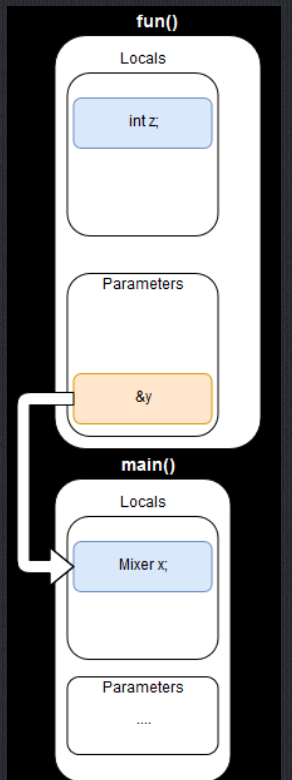
No copy elision

```
Mixer fun() {
    int z;
    Mixer x;
    return x;
}

Mixer y{fun()};
```



Copy elision

```
Mixer fun() {
    int z;
    Mixer x;
    return x;
}

Mixer y{fun()};
```



RVO in C++17

```
struct Factory {
  Factory(){};
  Factory(const Factory &) = delete;
  Factory(Factory &&) = delete;
};

Factory getFactory() { return Factory{}; }


// C++11/14
error: use of deleted function
       'Factory::Factory(Factory&&)'
```

"Guaranteed copy elision through simplified value categories"

- prvalue - initializer of object

- glvalue - location of object

- if prvalue is used as the initializer of an object with the same type, it initializes it directly

- consequence - when function returns object that is initialized by temporary, object is initialized directly, no copy/move op

- no copy/move ctor needed

Actually...

```
auto factory = getFactory(); // OK in C++17
```

- RVO only, NRVO works the old way

- it's not copy elision anymore

- no copy to elide
- deferred temporary materialization

When copy elision not eligible...

then create new object

copy or move

Move or copy?

```
Mixer fun1() {
    // easy - rvalue so move ctor
    return Mixer{};
}


Mixer fun2() {
    // but here's lvalue...
    Mixer x;
    return x;
}
```

C++11* criteria for RVO with move

32 When the criteria for elision of a copy operation are met or would be met save for the fact that the source object is a function parameter, and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If overload resolution fails, or if the type of the first parameter of the selected constructor is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as an lvalue. [*Note:* This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor to be called if elision is not performed, and the selected constructor must be accessible even if the call is elided. — *end note*]

So move instead of copy elision but criteria tied closely...

- when -fno-elide-constructors

- when decision not to perform copy elision as it's not mandatory

- when fun param

Let's recall examples with issues from the beginning

```
A)
unique_ptr<MixerInterface> create()
{
    unique_ptr<Mixer> mixer = make_unique<Mixer>();
    return mixer;
    // compiler error
}


G)
MixerInterface create()
{
    Mixer x;
    return x;
    // copy or move ctor
}
```

DR1579

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1579

2012-2014

1579. Return by converting move constructor

Section: 15.8 [class.copy] **Status:** C++14 **Submitter:** Jeffrey Yasskin **Date:** 2012-10-23

N3690 comment US 13

[Moved to DR at the February, 2014 meeting.]

Currently the conditions for moving from an object returned from a function are tied closely to the criteria for copy elision, which requires that the type of the object being returned be the same as the return type of the function. Another possibility that should be considered is to allow something like

```
optional<T> foo() {
  T t;
  ...
  return t;
}
```

and allow `optional<T>::optional(T&&)` to be used for the initialization of the return type. Currently this can be achieved explicitly by use of `std::move`, but it would be nice not to have to remember to do so.

DR1579 resolution

Proposed resolution (September, 2013):

Change 15.8 [class.copy] paragraph 32 as follows:

> When the criteria for elision of a copy operation are met ~~or would be met save for the fact that the source object is a function parameter,~~ and the object to be copied is designated by an lvalue, **or when the *expression* in a `return` statement is a (possibly parenthesized) *id-expression* that names an object with automatic storage duration declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*,** overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If overload resolution fails...

wording in C++14

added retrospectively to C++11 with C++14 standard publication

```
unique_ptr<MixerInterface> create()
{
    auto mixer = make_unique<Mixer>();
    return mixer;
}
```

pre DR1579 compiler reasoning:

- can I perform copy elision? No, types are different.

- let's create a new object on stack

- can I treat object as rvalue and try to use move ctor for this?

- no, criteria here are tied with copy elision.

- ctor overload resolution with lvalue

- dammit, only move converting ctor available, error.

```
unique_ptr<const MixerInterface> create()
{
    auto mixer = make_unique<MixerInterface>();
    return mixer;
}
```

post DR1579 compiler reasoning:

- can I perform copy elision? No, types are different.

- let's create a new object on stack

- can I treat object as rvalue and try to use move ctor for this?

- yes, it's an object with automatic storage duration, go ahead

- ctor overload successful, converting move ctor chosen

So why object slicing compiled on old compiler?

```
MixerInterface create()
{
    Mixer x;
    return x;
}
```

Return statement

- evaluates the expression

- terminates current function

- returns the result of the expression to the caller,

- after implicit conversion to the function return type.

```
MixerInterface create()
{
    Mixer x;
    return x;
}
```

pre DR1579 compiler reasoning:

- can I perform copy elision?
- no cause different types.
- let's create a new object on stack
- evaluate expression on return
- can I treat object as rvalue and try to use move ctor for this?

- no, since criteria are the same as for copy elision.
- so need to treat it as lvalue.
- implicit cast to return type
- ctor overload resolution - copy ctor

In unique_ptrs both decisions were also negative.
But no implicit cast nor copy ctor were available.

```
MixerInterface create()
{
    Mixer x;
    return x;
}
```

post DR1579 compiler reasoning:

- can I perform copy elision?
- no cause different types.
- let's create a new object on stack
- evaluate expression on return
- can I treat object as rvalue and try to use move ctor for this?

- yes, it's an object with automatic storage duration, go ahead
- but type of ctor arg is not rvalue ref to its param's type so I can't
- so need to treat as lvalue
- implicit cast to return type
- ctor overload resolution - copy ctor

Back to object slicing copy/move problem

Why gcc 8.1+ uses move?

```
MixerInterface create()
{
    Mixer x;
    return x;
    // gcc 8.1+      - move ctor
    // gcc 8.0-      - copy ctor
    // clang 8.0.0   - copy ctor
    // c++17 standard - copy ctor
}
```

gcc 8.1+ was ahead of C++17 standard.

Luckily there's C++20...

Current/future RVO optimization proposals

P0527
Implicitly move from rvalue references in return statements

David Stone

http://wg21.link/p0527

```
unique_ptr fun(unique_ptr && ptr) {
    return ptr;
}
```

error: use of deleted function 'unique_ptr::unique_ptr(const unique_ptr&)'

could be fixed by:

```
return move(ptr);
```

consider generic case

```
auto fun(args...) {
    decltype(auto) result = some_other_fun();
    return forward<decltype(result)>(result);
}
```

with std::forward, RVO inhibited when result is a value and not a reference

without std::forward, doesn't compile when result is rvalue ref and is move only

Surprising results

```
// move ctor
string f(std::string x) { return x; }

// copy ctor - quite surprising...
string g(string && x) { return x; }
```

- Similar suggestions regarding rvalue reference function parameters used in throw expression and exception declaration.

- Wording change to include rvalue reference:
"an object with automatic storage duration" -> "a movable entity"

Proposal included in C++20 standard.

P1155
More implicit moves

Arthur O'Dwyer
David Stone

http://wg21.link/p1155

Paper starts with:

"Programmers expect return x; to trigger copy elision; or, at worst, to implicitly move from x instead of copying. Occasionally, C++ violates their expectations and performs an expensive copy anyway. [...]"

Current iso cpp wording has issues

3 In the following copy-initialization contexts, a move operation might be used instead of a copy operation:

(3.1) — If the *expression* in a `return` statement (9.6.3) is a (possibly parenthesized) *id-expression* that names an object with automatic storage duration declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, or

(3.2) — if the operand of a *throw-expression* (8.5.17) is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing *try-block* (if there is one),

overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If the first overload resolution fails or was not performed, or if the type of the first parameter of the selected constructor is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as an lvalue. [*Note:* This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor to be called if elision is not performed, and the selected constructor must be accessible even if the call is elided. — *end note*]

- Throwing is pessimized

```
void seven(Widget w) {
    throw w;
}
```

Throw function parameter -
no copy elision, and no implicit move (the object is copied)

Note:

C++17 standard wording	- copy
GCC	- copy
Clang 4.0.1+, MSVC 2015+, ICC 16.0.3+	- implicit move

- Non-constructor conversion is pessimized

```
struct To {
    operator Widget() const &;
    operator Widget() &&;
};

Widget nine() {
    To t;
    return t;
}
```

Conversion operator instead of ctor -
no copy elision, and no implicit move (the object is copied)

- By-value sinks are pessimized

```
struct Fowl {
    Fowl(Widget);
};

Fowl eleven() {
    Widget w;
    return w;
}
```

Value instead of rvalue ref -
no copy elision, and no implicit move (the Widget object is copied)

Note:

C++17 standard wording	- copy
Clang, ICC, MSVC	- copy
GCC 5.1+	- implicit move

- Slicing is pessimized

```
Base thirteen() {
    Derived result;
    return result;
}
```

Different object's type -
no copy elision, and no implicit move (the object is copied)

Note:

C++17 standard wording	- copy
Clang, MSVC	- copy
GCC 8.1+, ICC 18.0.0+	- implicit move

"We propose to remove all four of these unnecessary limitations."

- implemented in clang in 02.2018

- -Wreturn-std-move (enabled as part of -Wmove, -Wmost, and -Wall)

- overload resolution done twice - standard rules and the ones proposed here

- if they differ, then diagnostic warning is emitted suggesting adding explicit std::move since copy otherwise

Feedback from production

Plenty of true positives

Proven with clang flag enabled on Chromium and LibreOffice projects

Lack of false positives

Just one complaint from Mozzilla saying that performance benefit was too little when compared to ugliness of explicit std::move. Actually it's not even mistake.

Both papers merged as P1825 and included in C++20

p0527

p1155

http://wg21.link/p1825

P0889
Ultimate copy elision

Antony Polukhin

http://wg21.link/p0889

A. Teaching practice

- code with numerous functions since we've been taught for decades to delegate to funs and classes

- compilers inline more aggressievly and inline a lot

B. Copy elision rules

- current rules for copy elision mostly assume that a function from source code remains a function in a binary

- inlining, aliasing reasoning, and link time optimization open a new world to possible optimizations since fun params and its body is seen together

- current rules are suboptimal for modern compilers: they prevent optimizations.

C. std::move/rvalue is not a panacea

move operation is not a solution since

std::array
std::basic_string
std::function
std::variant
std::optional

and others

may store a lot of data on stack

Summary

- year 2002 - the oldest proposal on move semantics found

Document number: N1377=02-0035

Howard E. Hinnant, hinnant@twcny.rr.com
Peter Dimov, pdimov@mmltd.net
Dave Abrahams, dave@boost-consulting.com

September 10, 2002

A Proposal to Add Move Semantics Support to the C++ Language

- Introduction
- Motivation
- Copy vs Move
- What is needed from the language?
- Binding temporaries to references
- More on A&&
- move_ptr Example
- Cast to rvalue

- move semantics present for almost 2 decades now
- there're still areas to optimize

The journey covered

- non compiling unique_ptr examples

- copy elision mechanics and rules

- RVO with move semantics

- changes and defects in C++11, C++14, C++17, C++20

- new proposals enhancing nr of contexts for copy elision and move object creation on fun return

Journey memoirs

- you can't blindly rely on what your compiler tells you

- upgrade compilers in your projects

- upgrade C++ standard

- less time on investigating bugs

- increased performance

- cleaner codebase

References

[00] David Stone - P0527R1 Implicitly move from rvalue references in return statements
[01] Arthur O'Dwyer, David Stone - P1155R2 More implicit moves
[02] Antony Polukhin - P0889R1 Ultimate copy elision
[03] Arthur O'Dwyer - RVO harder than it looks - CppCon 2018
[04] John Kalb - Copy elision - CppCon 2018
[05] Jonas Devlieghere - Guaranteed Copy Elision
[06] Simon Brand - Guaranteed Copy Elision Does Not Elide Copies
[07] David Rodriguez - Value semantics: Copy elision
[08] David Rodriguez - Value semantics: NRVO
[09] Zhao Wu IBM - RVO V.S. std::move

[10] Scott Meyers - Effective Modern C++
[11] A proposal to add move semantics support to the C++ Language - 2002
[12] C++11 RVO defect report
[13] cppreference - return statement
[14] cppreference - copy elision
[15] cppreference - value category

Thank you

Lukasz Bondyra

Senior Engineer @ Motorola Solutions

https://github.com/bondyr/rvo_ppt

Lkbondyra@gmail.com