Return Value Optimization

Harder Than It Looks

Arthur O'Dwyer
2018-09-25

Outline

- What is the “return slot”? [3–34]
- Rules of thumb for RVO [35–37]
- How RVO *actually* works in standard C++ [38–45]
- Test cases and war stories [46–54]
- Writing a new Clang diagnostic... [55–69]
- ...and a standards proposal (P1155) [70–80]

Hey look!
Slide numbers!

What is the “return slot”?

```
int apple()
{
    return 42;
}

int pear()
{
    return 1 + apple();
}
```

Let’s start with the simplest possible function.

x86-64 calling convention

```
int apple()
{
    return 42;
}

int pear()
{
    return 1 + apple();
}
```

clang -O0 -fomit-frame-pointer -S

```
_Z5applev:
  movl $42, %eax
  retq

_Z4pearv:
  callq _Z5applev
  addl $1, %eax
  retq
```

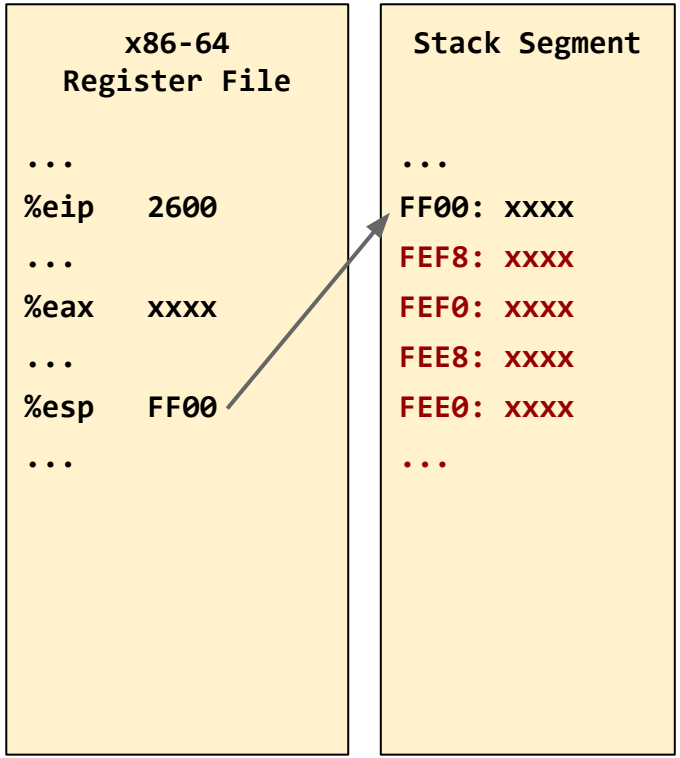
On x86-64, the function's return value usually goes into the %eax register.

x86-64 calling convention

```
int apple()
{
  return 42;
}

int pear()
{
  return 1 +
    apple();
}
```

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq

2600:             callq _Z4pearv
2605:             do something else
```

eip = 2600

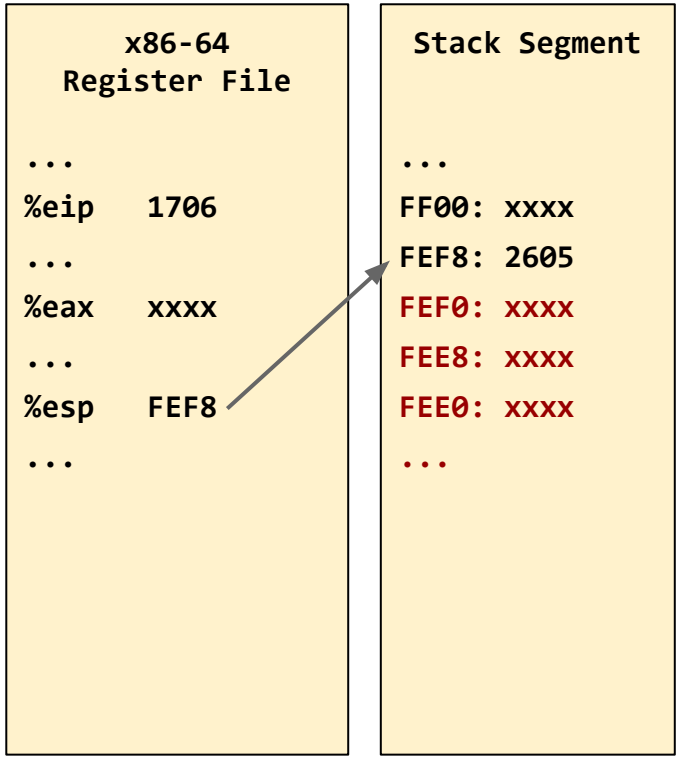


x86-64 calling convention

```
int apple()
{
  return 42;
}

int
{
  return 1 +
    apple();
}
```

eip = 1706

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq


2600:             callq _Z4pearv
2605:             do something else
```

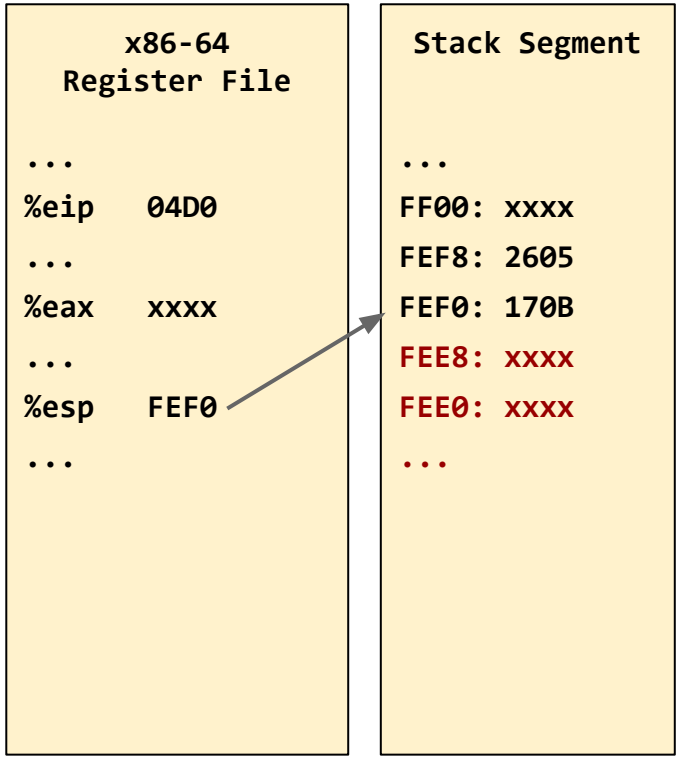


x86-64 calling convention

```
int apple()
{
  r
}

int pear()
{
  return 1 +
    apple();
}
```

eip = 04D0

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq

2600:             callq _Z4pearv
2605:             do something else
```

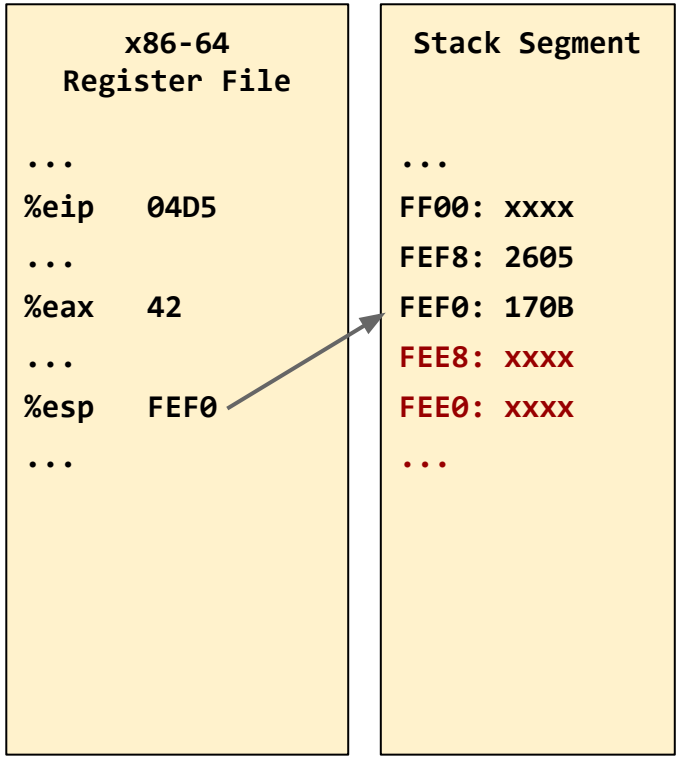


x86-64 calling convention

```
int apple()
{
  return 42;
}

int pear()
{
  return 1 +
    apple();
}
```

eip = 04D5

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq


2600:             callq _Z4pearv
2605:             do something else
```

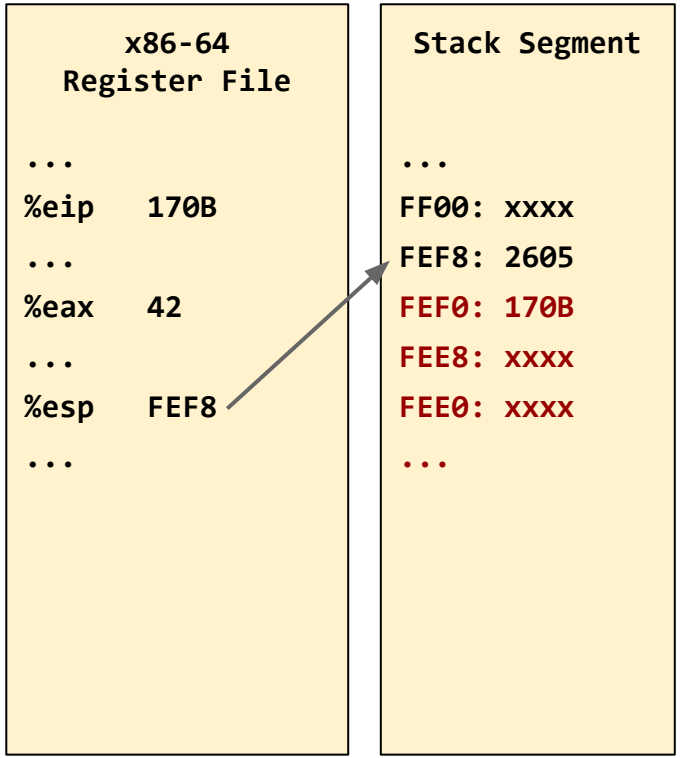


x86-64 calling convention

```
int apple()
{
  return 42;
}

int pear()
{
  return 1 +
    apple();
}
```

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq


2600:             callq _Z4pearv
2605:             do something else
```

eip = 170B

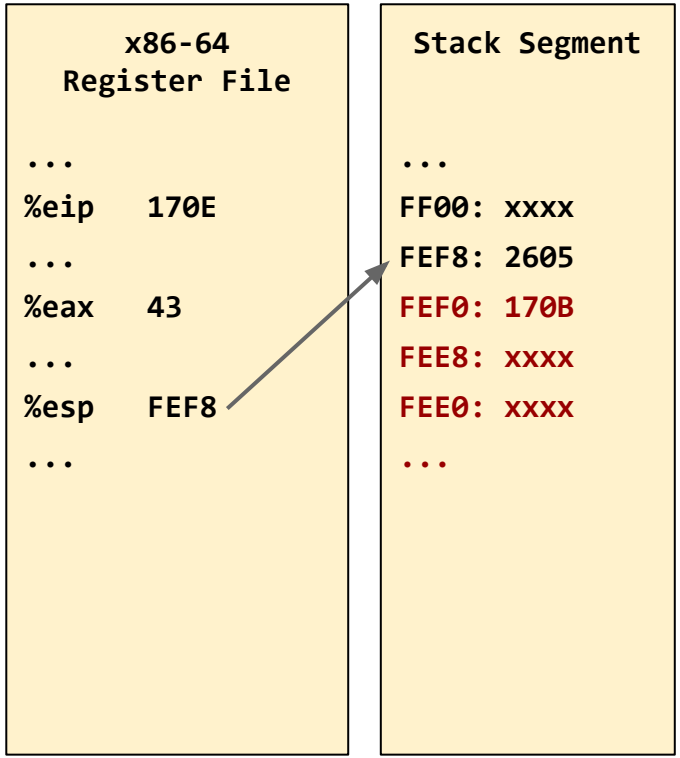


x86-64 calling convention

```
int apple()
{
  return 42;
}

int pear()
{
  r
    apple();
}
```

eip = 170E

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq


2600:             callq _Z4pearv
2605:             do something else
```

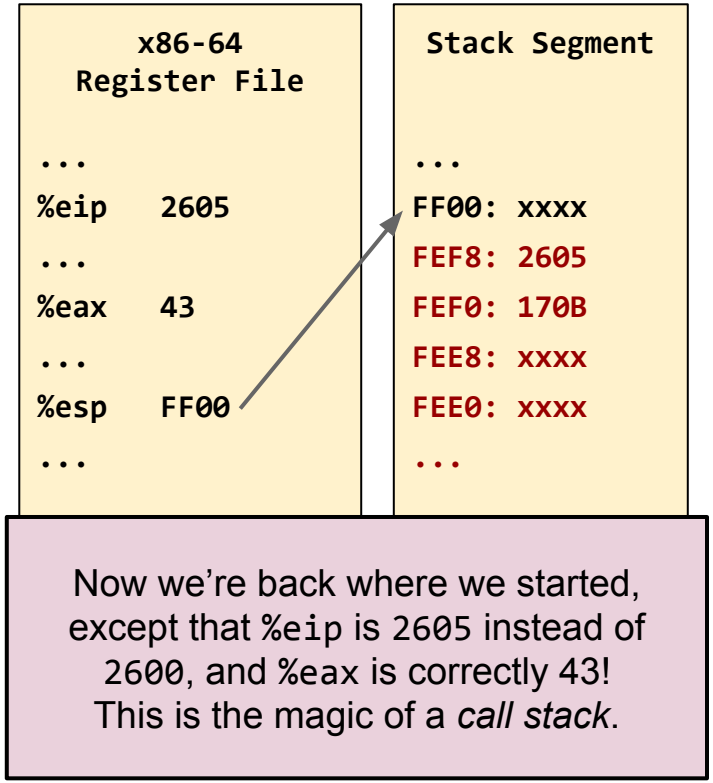


x86-64 calling convention

```
int apple()
{
  return 42;
}

int pear()
{
  return 1 +
    apple();
}
```

```
                _Z5applev:
04d0:             movl $42, %eax
04d5:             retq

                _Z4pearv:
1706:             callq _Z5applev
170b:             addl $1, %eax
170e:             retq

2600:             callq _Z4pearv
2605:             do something else
```

eip = 2605



Now we're back where we started, except that %eip is 2605 instead of 2600, and %eax is correctly 43! This is the magic of a *call stack*.

But what do we do with big objects?

```
struct Fruit {
    int data[5];  // 20 bytes
};

Fruit apple()
{
    return Fruit{1,2,3,4,5};  // Can’t return this in %eax!
}


... apple() ...
```

But what do we do with big objects?

```
struct Fruit {
    int data[5];  // 20 bytes
};

Fruit apple()
{
    return Fruit{1,2,3,4,5};
}


... apple() ...
```

```
clang -O0 -fomit-frame-pointer -S

_Z5applev:
  movq %rdi, %rax
  movl $1, (%rdi)
  movl $2, 4(%rdi)
  movl $3, 8(%rdi)
  movl $4, 12(%rdi)
  movl $5, 16(%rdi)
  retq

...
  subq $24, %rsp
  movq %rsp, %rdi
  callq _Z5applev
  # now %rsp[0..20] hold our answer
...
```

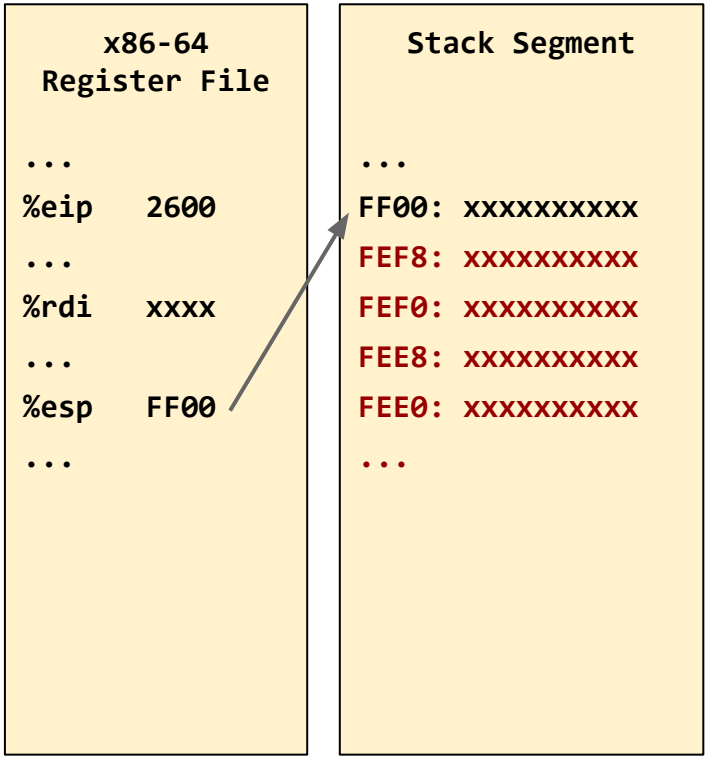
For big objects, the compiler adds a hidden function parameter: a "return slot address," passed in `%rdi`.

x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  ap    ()
...
```

eip = 2600

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

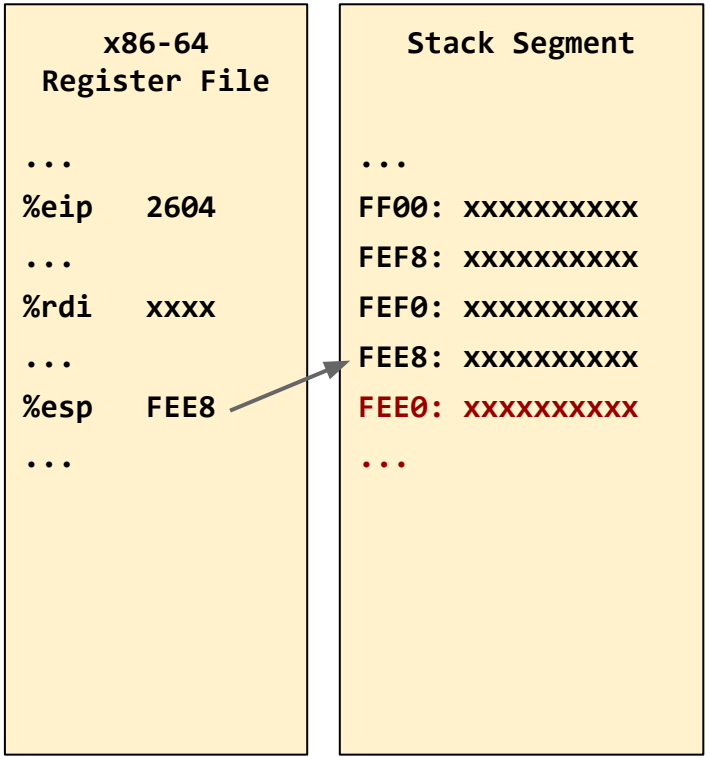


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  ap
...
```

eip = 2604

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

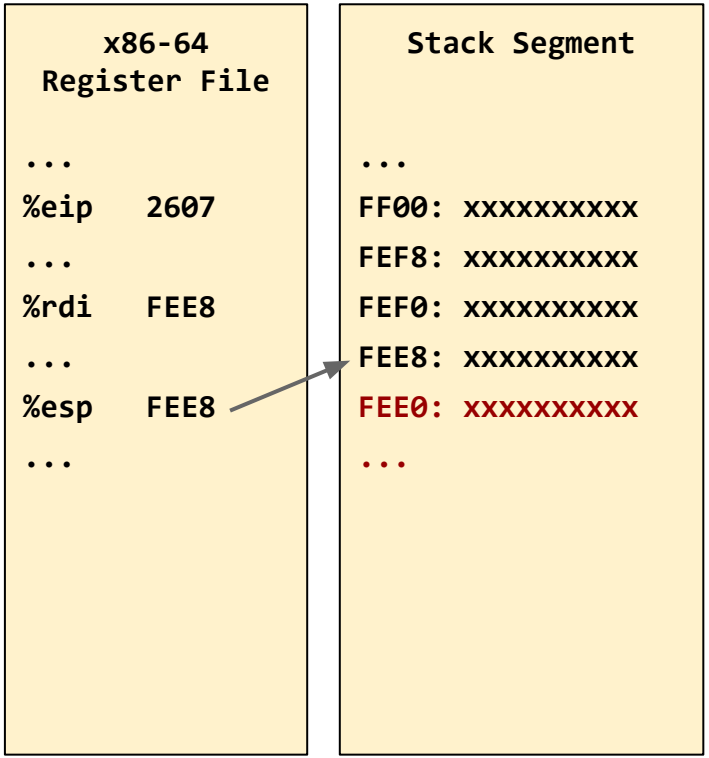


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 2607

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

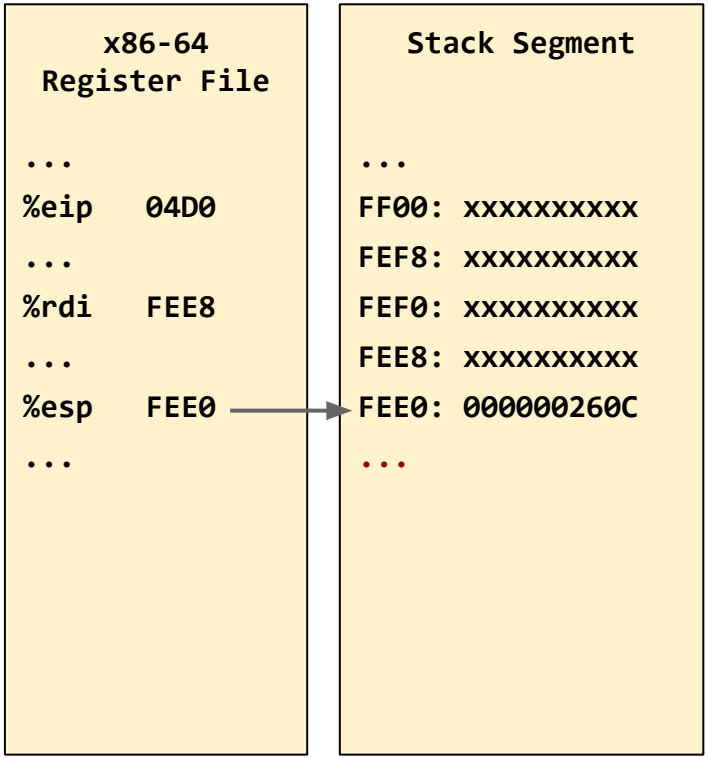


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 04D0

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

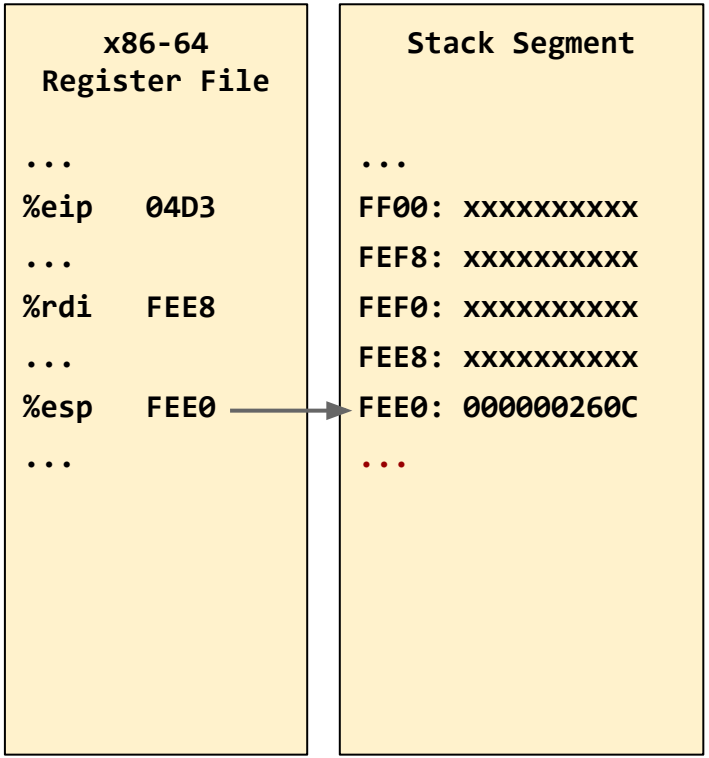


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 04D3

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

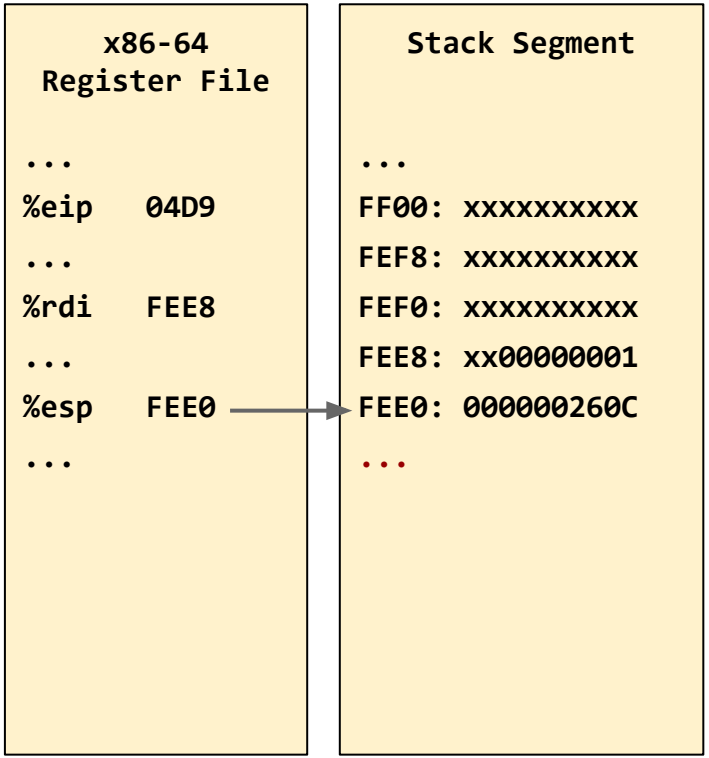


x86-64 conventions for big objects

```
Fruit apple()
{
  re
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 04D9

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

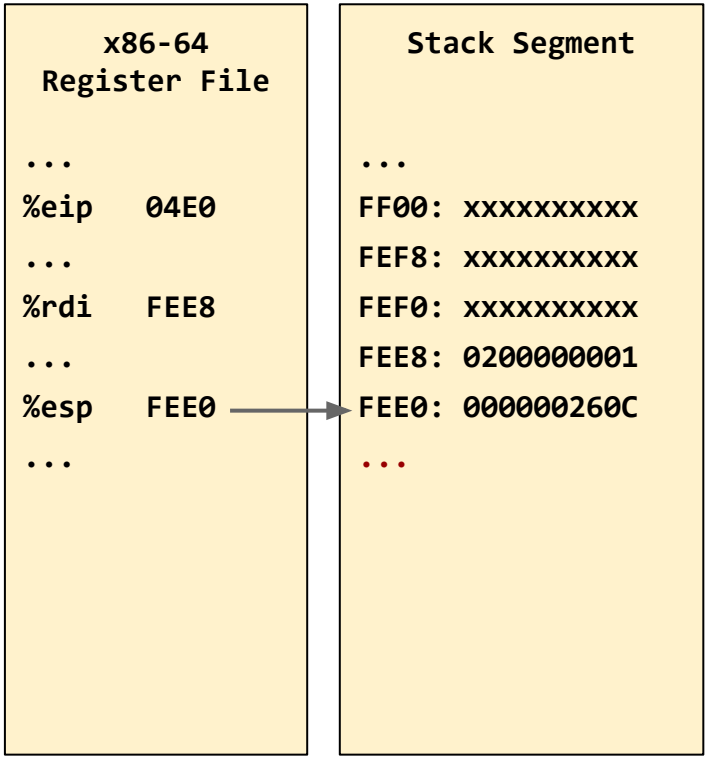


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
  };
}

...
  apple()
...
```

eip = 04E0

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

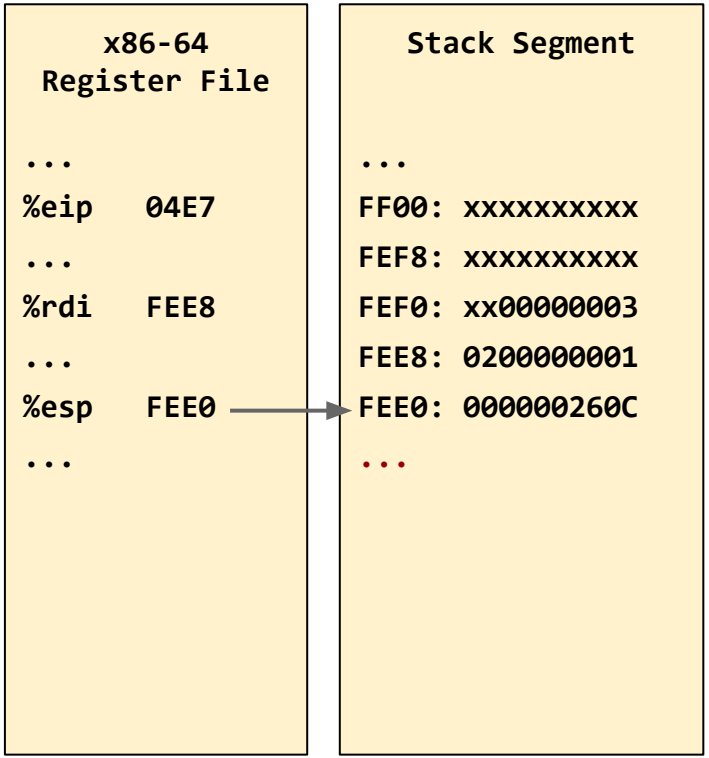


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1, 2, 3, 4, 5
  };
}

...
  apple()
...
```

eip = 04E7

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

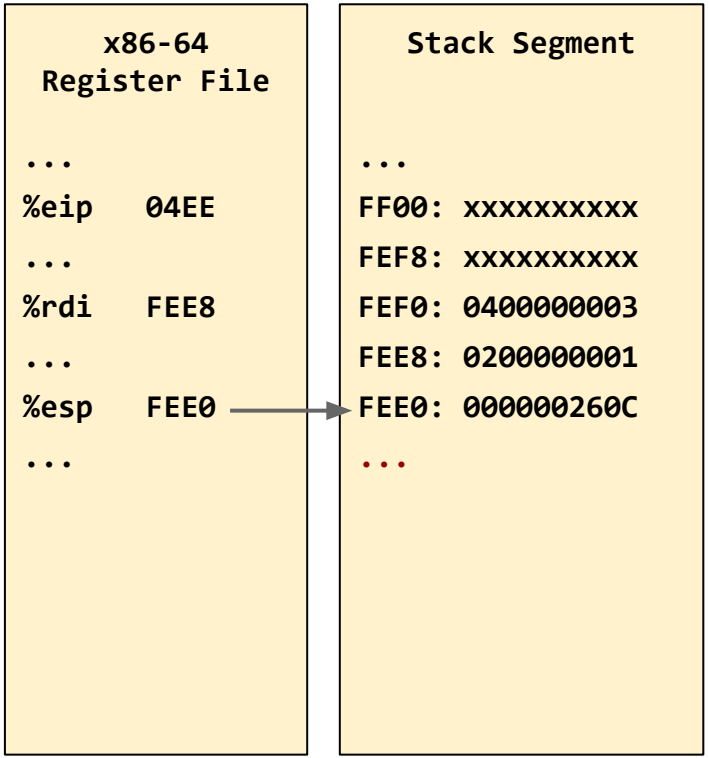


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 04EE

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

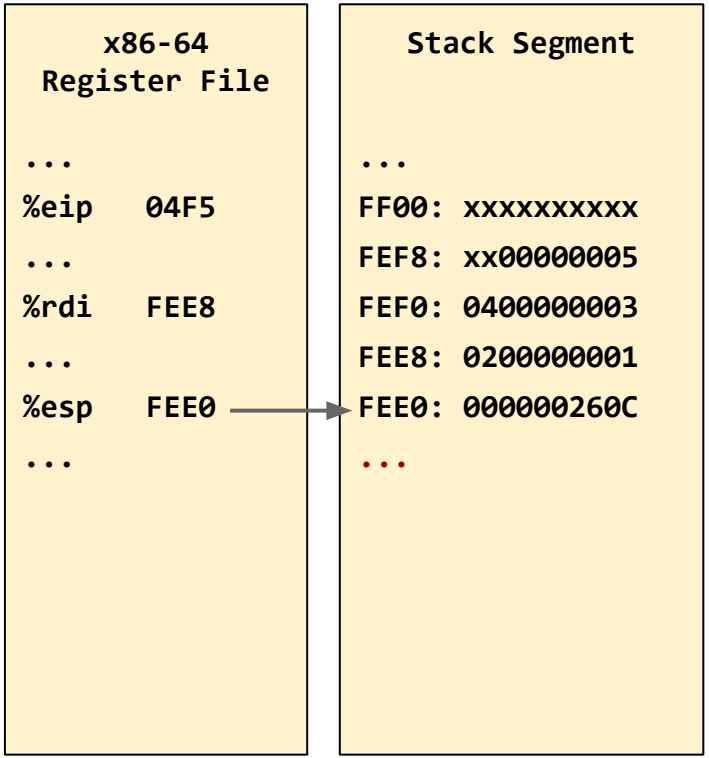


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 04F5

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

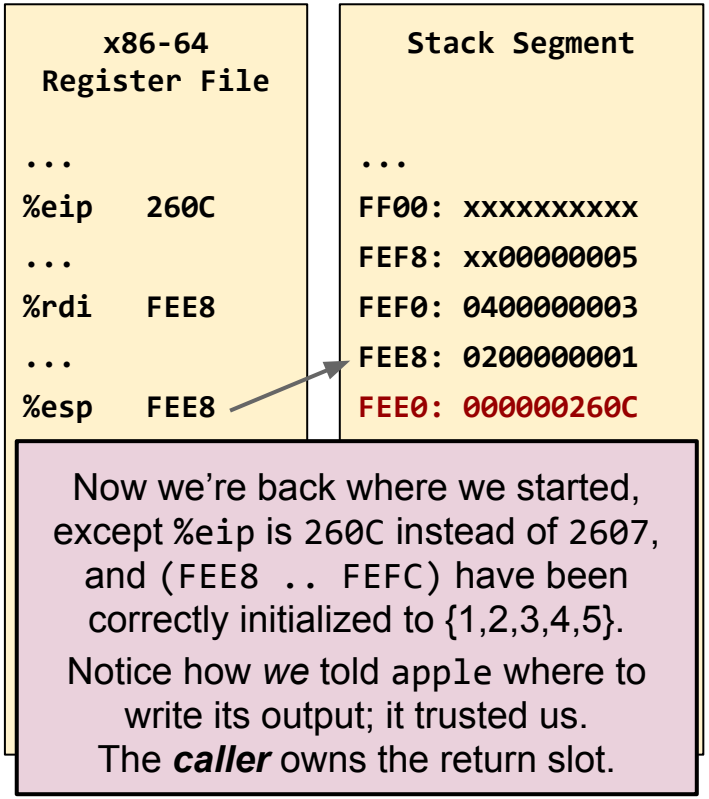


x86-64 conventions for big objects

```
Fruit apple()
{
  return Fruit{
    1,2,3,4,5
  };
}

...
  apple()
...
```

eip = 260C

```
   _Z5applev:
04d0:  movq %rdi, %rax
04d3:  movl $1, (%rdi)
04d9:  movl $2, 4(%rdi)
04e0:  movl $3, 8(%rdi)
04e7:  movl $4, 12(%rdi)
04ee:  movl $5, 16(%rdi)
04f5:  retq

   ...
2600:  subq $24, %rsp
2604:  movq %rsp, %rdi
2607:  callq _Z5applev
260c:  # examine %rsp[0..20]
   ...
```

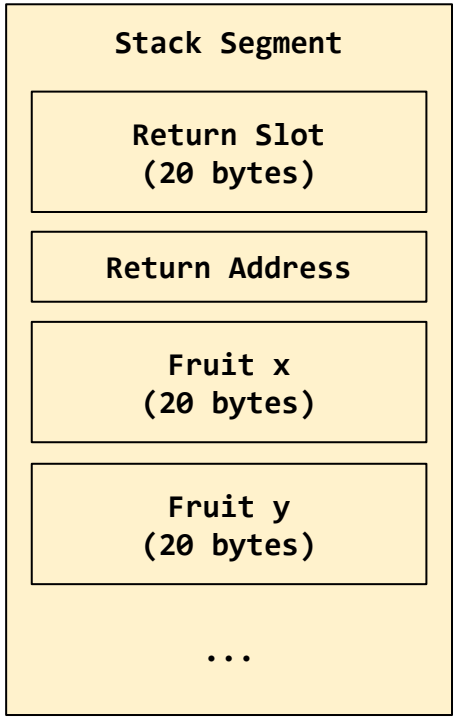


Moving into the return slot

```
struct Fruit {
    int data[5];
    Fruit(Fruit&&);
};

Fruit apples_and_oranges(bool condition)
{
    Fruit x = ...;
    Fruit y = ...;
    return std::move(condition ? x : y);
}
```

Moving into the return slot

```
struct Fruit {
    int data[5];
    Fruit(Fruit&&);
};

Fruit apples_and_oranges(bool condition)
{
    Fruit x = ...;
    Fruit y = ...;
    return std::move(condition ? x : y);
}
```



Copy elision

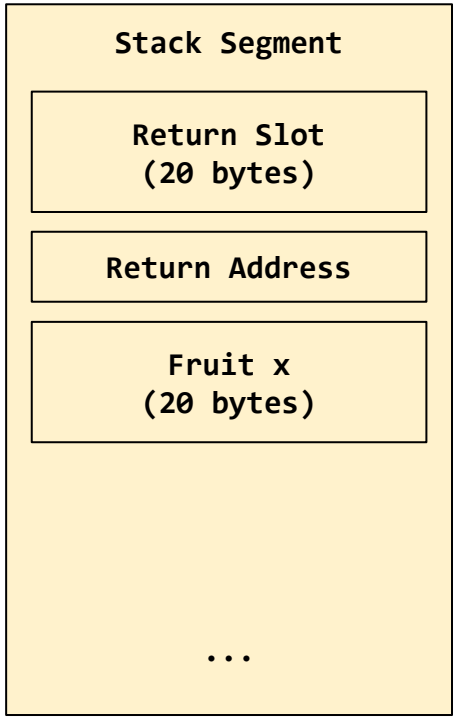
```
struct Fruit {
    int data[5];
};

Fruit nothing_but_apples()
{
    Fruit x = ...;
    return x;
}
```

Copy elision

```
struct Fruit {
    int data[5];
};

Fruit nothing_but_apples()
{
    Fruit x = ...;
    return x;
}
```

```
_Z18nothing_but_applesv:
  subq $24, %rsp
  movq %rdi, %rax
  movl $1, (%rsp)
  movl $2, 4(%rsp)
  movl $3, 8(%rsp)
  movl $4, 12(%rsp)
  movl $5, 16(%rsp)
  movq (%rsp), (%rdi)
  movq 8(%rsp), 8(%rdi)
  movl 16(%rsp), 16(%rdi)
  addq $24, %rsp
  retq
```

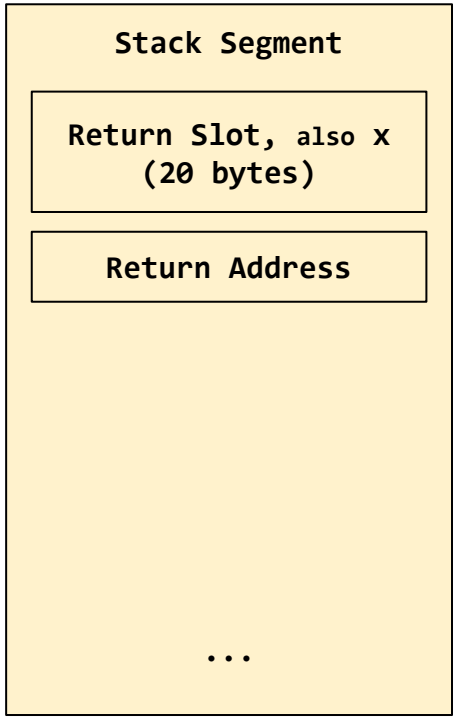
This code (slightly altered for presentation) constructs `Fruit x` at `%rsp[0..20]` and then copies it into the return slot at `%rdi[0..20]`.



Copy elision

```
struct Fruit {
    int data[5];
};

Fruit nothing_but_apples()
{
    Fruit x = ...;
    return x;
}
```

```
_Z18nothing_but_applesv:
  movq %rdi, %rax
  movl $1, (%rdi)
  movl $2, 4(%rdi)
  movl $3, 8(%rdi)
  movl $4, 12(%rdi)
  movl $5, 16(%rdi)
  retq
```



But we can do better! Here we construct `Fruit x` directly in the return slot at `%rdi[0..20]`.

Copy elision rules in C++

- In C++03 through C++14, copy elision was *permitted* in many circumstances, which we'll discuss in a moment.
- In C++17, copy elision became *mandatory* in many circumstances.
- But there are cases where copy elision is not always possible...

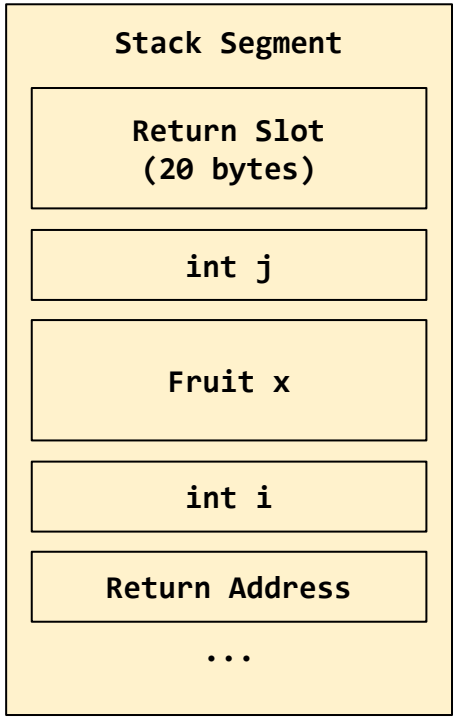
When can’t we elide?

```
Fruit apples_to_apples(int i, Fruit x, int j)
{
    return x;
}
```

In this (slightly altered) example, the caller passes in `Fruit x` at one stack address, and the return slot at a different stack address.

We must get the data out of `x` and into the return slot somehow!

We can’t elide the copy because we don’t control `x`’s physical location.

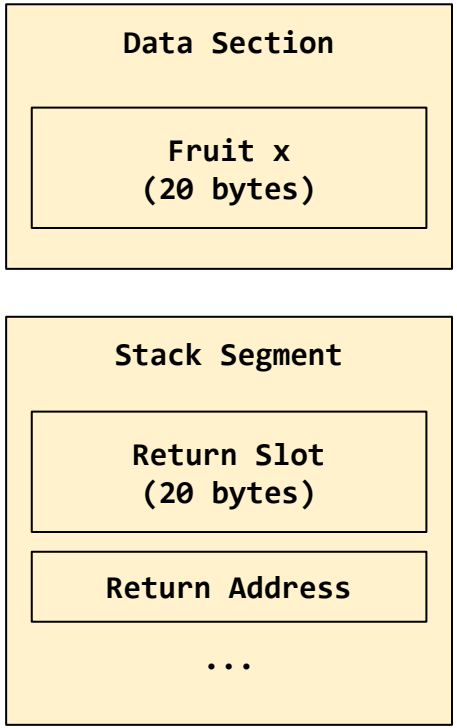


When can’t we elide?

```
static Fruit x;

Fruit apples_to_apples()
{
    return x;
}
```

Reductio ad absurdum:

We can’t elide this copy because we don’t control x’s physical location.

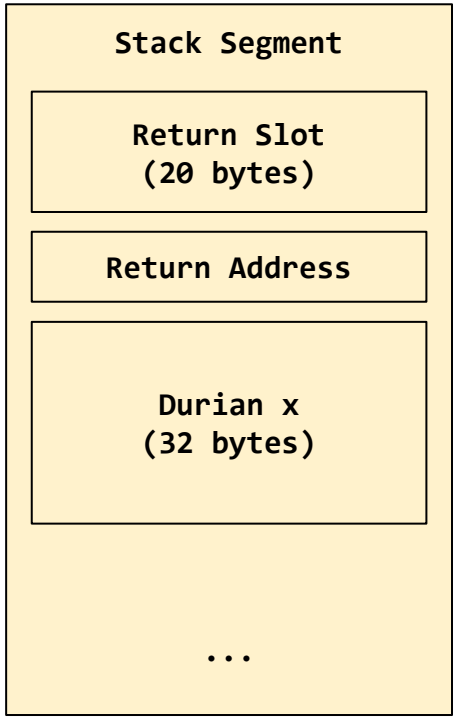


One more important case...

Slicing to base class

```
struct Durian : Fruit {
    double smell;
};

Fruit slapchop()
{
    Durian x = ...;
    return x;
}
```

We can't elide this copy because, while we do control x's physical location, x is of the "wrong" type for constructing into the return slot.



Rules of thumb for RVO

Here's what I used to tell people when they asked about RVO:

- "Unnamed RVO" (URVO): Returning a temporary (a prvalue) will trigger copy-elision.

```
return Fruit{1,2,3,4,5};
return my_helper_function();
```

- "Named RVO" (NRVO): Returning a local variable "by name" will trigger copy-elision, except in the corner cases we've covered.

```
return x;
```

- And even if copy-elision *doesn't* happen, *implicit move* happens...

Rules of thumb for RVO

Here's what I used to tell people when they asked about RVO:

- "Implicit move": When returning a local variable "by name" doesn't trigger copy-elision, the compiler's overload resolution will still automatically treat the name x as an rvalue!

```
std::string identity(std::string x) {
    return x;  // x will be implicitly moved-from, not copied!
}
```

- Because of C++11's "implicit move," writing `return std::move(x)` is almost always a pessimization — it never helps, and it might hurt by disabling NRVO.


I WANT THE TRUTH!

Story time!

```
extern "C" {
    typedef enum { CEK_NONE, CEK_INT, CEK_STRING, CEK_LIST } cexpr_kind;
    typedef struct cexpr cexpr;

    cexpr *cexpr_new_int(int value);
    cexpr *cexpr_new_string(const char *data);
    cexpr *cexpr_new_empty_list();
    cexpr *cexpr_clone(cexpr *self);
    cexpr *cexpr_free(cexpr *self);

    cexpr_kind cexpr_get_kind(cexpr *self);
    int cexpr_int_value(cexpr *self);
    int cexpr_list_length(cexpr *self);
    void cexpr_list_append(cexpr *self, cexpr *v);
}
```

C++ wrapper around C API

```
class Cexpr {
protected:
    cexpr *impl_;
    Cexpr(cexpr *p) : impl_(p) {}
public:
    Cexpr(const Cexpr& rhs) : impl_(cexpr_clone(rhs.impl_)) {}
    Cexpr(Cexpr&& rhs) : impl_(rhs.impl_) { rhs.impl_ = nullptr; }
    ~Cexpr() { cexpr_free(impl_); }
};

class CexprInt : public Cexpr {
public:
    CexprInt(int value) : impl_(cexpr_new_int(value)) {}
    int value() const { return cexpr_int_value(impl_); }
};
```

C++ wrapper around C API

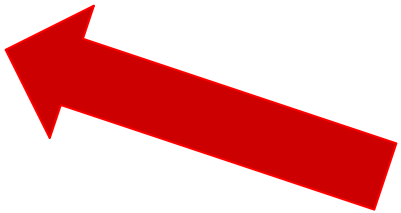
```
class CexprList : public Cexpr {
public:
    CexprList() : impl_(cexpr_new_empty_list()) {}
    int length() const { return cexpr_list_length(impl_); }

    void append(const Cexpr& v) {
        cexpr *p = cexpr_clone(v.impl_);
        cexpr_list_append(impl_, p);
    }

    void append(Cexpr&& v) {
        cexpr *p = std::exchange(v.impl_, nullptr);
        cexpr_list_append(impl_, p);
    }
};
```

C++ functions that return Cexpr

```
class ConfigManager {
    virtual Cexpr as_cexpr() const = 0;
};

class ConfigManagerImpl : public ConfigManager {
    Cexpr as_cexpr() const override {
        CexprList cfg;
        cfg.append(CexprInt(17));
        cfg.append(CexprInt(42));
        cfg.append(CexprString("hike"));
        return cfg;
    }
};
```

C++ functions that return Cexpr

```
class ConfigManager {
    virtual Cexpr as_cexpr() const = 0;
};

class ConfigManagerImpl : public ConfigManager {
    Cexpr as_cexpr() const override {
        CexprList cfg;
        cfg.append(CexprInt(17));
        cfg.append(CexprInt(42));
        cfg.append(CexprString("hike"));
        return cfg;
    }
};
```

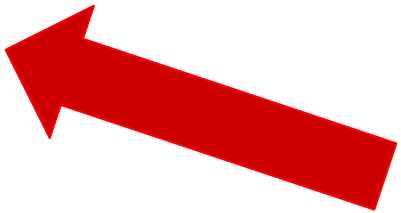


Now QA determines that such-and-such an operation is taking much longer than it should. We profile the code and discover that we're spending an awful lot of cycles in `cexpr_clone()`.
(Not literally in this config code, but somewhere using exactly the same pattern of construction and return.)

C++ functions that return Cexpr

```
class ConfigManager {
    virtual Cexpr as_cexpr() const = 0;
};

class ConfigManagerImpl : public ConfigManager {
    Cexpr as_cexpr() const override {
        CexprList cfg;
        cfg.append(CexprInt(17));
        cfg.append(CexprInt(42));
        cfg.append(CexprString("hike"));
        return cfg;
    }
};
```

Now QA determines that such-and-such an operation is taking much longer than it should. We profile the code and discover that we're spending an awful lot of cycles in `cexpr_clone()`.
(Not literally in this config code, but somewhere using exactly the same pattern of construction and return.)



"But you told me that `return x` would treat x as an rvalue! Why is it copying?"

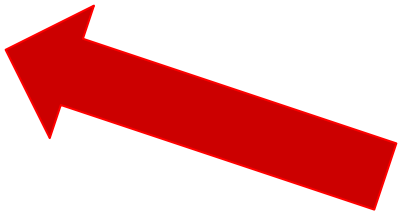
The actual rules of “implicit move”

Before 2013, the rules were slightly different — but even the C++11 rules were posthumously amended by Defect Report CWG1579.

> [class.copy.elision] /3 If 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* [... then ...] 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.

What went wrong here? Slicing.

```
Cexpr as_cexpr() const override {
    CexprList cfg;
    cfg.append(CexprInt(17));
    cfg.append(CexprInt(42));
    cfg.append(CexprString("hike"));
    return cfg;
}
```



The constructor that we want to call here is `Cexpr(Cexpr&&)`.

But `cfg`'s type is `CexprList`, not `Cexpr`!

"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."

The slicing here is 100% intentional; but it still silently disables the optimization we thought the compiler was giving us.

War story #2: `inplace_function`

```
template<class R, class... Args, size_t Capacity, size_t Alignment>
class inplace_function<R(Args...), Capacity, Alignment>
{
    vtable_t *vptr_;
    alignas(Alignment) char storage_[Capacity];
public:
    template<size_t SrcCap, size_t SrcAlign>
    inplace_function(inplace_function<R(Args...), SrcCap, SrcAlign> const & rhs)
        requires(Capacity >= SrcCap && Alignment >= SrcAlign)
    {
        vptr_ = rhs.vptr_;
        vptr_->copy(storage_, rhs.storage_);
    }
    template<size_t SrcCap, size_t SrcAlign>
    inplace_function(inplace_function<R(Args...), SrcCap, SrcAlign>&& rhs)
        requires(Capacity >= SrcCap && Alignment >= SrcAlign)
    {
        vptr_ = std::exchange(rhs.vptr_, empty_function_vptr);
        vptr_->move(storage_, rhs.storage_);
    }
};
```

"Converting constructor": I know how to construct myself from the source type, stealing his guts if necessary.

Maintainer proposed this instead:

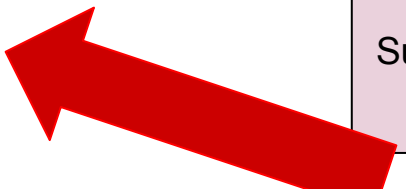
```
template<class R, class... Args, size_t Capacity, size_t Alignment>
class inplace_function<R(Args...), Capacity, Alignment>
{
    inplace_function(auto vptr, auto func, void *rhs_storage) : vptr_(vptr) {
        func(storage_, rhs_storage);
    }
public:
    template<size_t DstCap, size_t DstAlign>
    operator inplace_function<R(Args...), DstCap, DstAlign>() const &
        requires(DstCap >= Capacity && DstAlign >= Alignment)
    {
        return {vptr_, vptr_->copy, storage_};
    }
    template<size_t DstCap, size_t DstAlign>
    operator inplace_function<R(Args...), DstCap, DstAlign>() &&
        requires(DstCap >= Capacity && DstAlign >= Alignment)
    {
        auto vptr = std::exchange(vptr_, empty_function_vptr);
        return {vptr, vptr->move, storage_};
    }
};
```

"Conversion operator": I know how to convert myself to the destination type, lending him my own guts if necessary.

This seems to match our mental model a bit better.

What happens here?

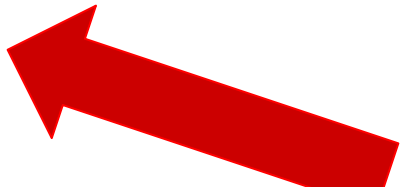
```
using IPF32 = inplace_function<void(), 32>;
using IPF64 = inplace_function<void(), 64>;

IPF32 get_upstream_callback() {
    return [msg = std::string(...)](){
        std::cout << msg << std::endl;
    };
}

IPF64 ConfigManagerImpl::callback() {
    auto result = get_upstream_callback();
    assert(!result.empty());
    return result;
}
```



Suddenly we're spending a lot of time in `std::string`'s copy constructor!

What went wrong here? Converting.

```
using IPF32 = inplace_function<void(), 32>;
using IPF64 = inplace_function<void(), 64>;

IPF32 get_upstream_callback() {
    return [msg = std::string(...)](){
        std::cout << msg << std::endl;
    };
}

IPF64 ConfigManagerImpl::callback() {
    auto result = get_upstream_callback();
    assert(!result.empty());
    return result;
}
```



Here we convert `IPF32` to `IPF64`.

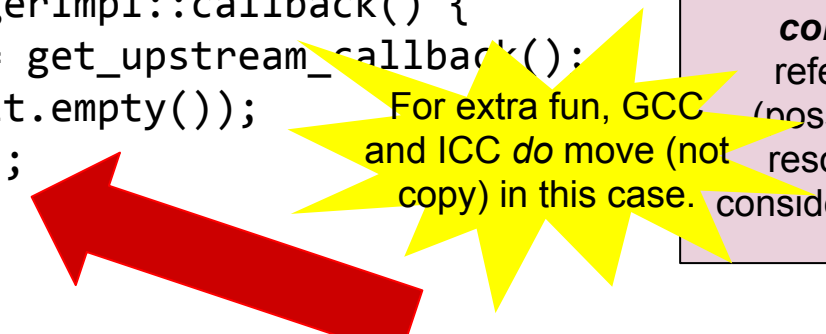
This works fine before the patch: overload resolution finds `IPF64::IPF64(IPF32&&)`.

After the patch, it finds `IPF32::operator IPF64()&&` instead.

"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."

What went wrong here? Converting.

```
using IPF32 = inplace_function<void(), 32>;
using IPF64 = inplace_function<void(), 64>;

IPF32 get_upstream_callback() {
    return [msg = std::string(...)](){
        std::cout << msg << std::endl;
    };
}

IPF64 ConfigManagerImpl::callback() {
    auto result = get_upstream_callback();
    assert(!result.empty());
    return result;
}
```



Here we convert `IPF32` to `IPF64`.

This works fine before the patch: overload resolution finds `IPF64::IPF64(IPF32&&)`.

After the patch, it finds `IPF32::operator IPF64()&&` instead.

"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."

Why are *constructors* so important?

Why does [class.copy.elision] /3 put such emphasis on finding constructors that take ***rvalue references to exact class types?*** What's so important about this particular special case that they decided to bake it into the standard? And not even bake it into C++14, but actually retroactively insert the same rule into C++11!?

Why are *constructors* so important?

Why does [class.copy.elision] /3 put such emphasis on finding constructors that take ***rvalue references to exact class types?*** What's so important about this particular special case that they decided to bake it into the standard? And not even bake it into C++14, but actually retroactively insert the same rule into C++11!?

```
std::unique_ptr<ConfigManager> create() {
    auto p = std::make_unique<ConfigManagerImpl>();
    return p;
}
```

Why are *constructors* so important?

Why does [class.copy.elision] /3 put such emphasis on finding constructors that take ***rvalue references to exact class types?*** What's so important about this particular special case that they decided to bake it into the standard? And not even bake it into C++14, but actually retroactively insert the same rule into C++11!?

The C++ Standard Library *looooves* move-enabled converting constructors.

```
unique_ptr<T>(unique_ptr<U>&&)      optional<T>(T&&)
shared_ptr<T>(shared_ptr<U>&&)      Expected<T>(T&&)
function<Sig>(Lambda&&)             variant<T,U>(T&&)
```

All of these examples use explicitly declared converting constructors. So when they dealt with this issue in the core language, they didn't think about any of the other ways you could get implicit conversion (slicing, conversion operators, or combinations of these).

Automatically detect problem cases

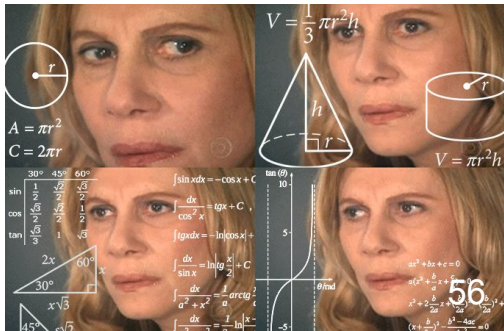
- Can we get the compiler to warn us when we write `return x` but `return std::move(x)` would have been more efficient?
- We can definitely do the opposite. **`clang++ -Wmove`** will warn about places we wrote `return std::move(x)` when `return x` would have been more efficient.

```
test.cc:8:12: warning: moving a local object in a return statement
      prevents copy elision [-Wpessimizing-move]
    return std::move(x);
           ^
test.cc:8:12: note: remove std::move call here
    return std::move(x);
           ^~~~~~~~~~ ~
```

Automatically detect problem cases

- Can we get the compiler to warn us when we write `return x` but `return std::move(x)` would have been more efficient?
- We can definitely do the opposite. **`clang++ -Wmove`** will warn about places we wrote `return std::move(x)` when `return x` would have been more efficient.

```
test.cc:8:12: warning: moving a local object in a return statement
      prevents copy elision [-Wpessimizing-move]
    return std::move(x);
           ^
test.cc:8:12: note: remove std::move call here
    return std::move(x);
           ^~~~~~~~~~ ~
```



Automatically detect problem cases

- As January 2018, Clang did ***not*** have a warning for the opposite, sneakier case, when we wrote `return x` expecting a move but got a copy instead.
- So I went and added one!

Automatically detect problem cases

First I tried adding a check to clang-tidy.

This great blog post got me off to a good start:
http://bbannier.github.io/blog/2015/05/02/Writing-a-basic-clang-static-analysis-check.html

Protip: `git clone --depth=1 http://llvm.org/git/llvm.git`

But clang-tidy is basically just for syntactic issues. It can do simple things such as observing that an implicit conversion has resolved to `Foo(const Bar&)` when `Foo(Bar&&)` exists, but it cannot “re-evaluate” the entire overload resolution step to see whether `Foo(Bar&&)` would actually have been a better match, or whether `Bar::operator Foo()&&` would have been selected, etc.

So I abandoned clang-tidy and went straight for a full-blown clang diagnostic.

Automatically detect problem cases

Adding the new diagnostic was easy because I could look at existing commits and pull requests that added new diagnostics, and copy them.

Especially D7633 which added `-Wpessimizing-move`.

- Step 1: Add the machinery for the command-line option.
- Step 2: Add a test case.
- Step 3: Actually add the new code to `libSema`.

Step 1: Add command-line option

```
+++ b/include/clang/Basic/DiagnosticGroups.td
@@ -380,7 +380,11 @@
 def ExplicitInitializeCall : DiagGroup<"explicit-initialize-call">;
 def Packed : DiagGroup<"packed">;
 def Padded : DiagGroup<"padded">;
 def PessimizingMove : DiagGroup<"pessimizing-move">;
+def ReturnStdMove : DiagGroup<"return-std-move">;
 def PointerArith : DiagGroup<"pointer-arith">;
 def PoundWarning : DiagGroup<"#warnings">;
 def PoundPragmaMessage : DiagGroup<"#pragma-messages">,
```

Step 1: Add command-line option

```
+++ b/include/clang/Basic/DiagnosticSemaKinds.td
@@ -5626,6 +5626,19 @@
 def note_remove_move : Note<"remove std::move call here">;

+def warn_return_std_move : Warning<
+  "local variable %0 will be copied despite being "
+  "%select{returned|thrown}1 by name">,
+  InGroup<ReturnStdMove>, DefaultIgnore;
+def note_add_std_move : Note<
+  "call 'std::move' explicitly to avoid copying">;
+
 def warn_string_plus_int : Warning<
   "adding %0 to a string does not append to the string">,
   InGroup<StringPlusInt>;
```

It took me several iterations to tease out the syntax of the diagnostic engine's mini-language.

Notice that the number "1" goes *outside* these curly braces.

Step 2: Add the test case

```
+++ b/test/SemaCXX/warn-return-std-move.cpp
@@ -0,0 +1,327 @@
+// RUN: %clang_cc1 -fsyntax-only -Wreturn-std-move -std=c++11 -verify %s
+// RUN: %clang_cc1 -fsyntax-only -Wreturn-std-move -std=c++11 \
    -fdiagnostics-parseable-fixits %s 2>&1 | FileCheck %s
+
+struct Base { stuff };
+struct Derived : public Base {};
+
+Base test() {
+    Derived d2;
+    return d2;  // e1
+    // expected-warning@-1{{will be copied despite being returned by name}}
+    // expected-note@-2{{to avoid copying}}
+    // CHECK: fix-it:"{{.*}}":{[[@LINE-3]]:12-[[@LINE-3]]:14}:"std::move(d2)"
+}
+
+and so on
```

The comment `// e1` is just for my own benefit when compiling the test by hand: I number the expected warnings e1, e2, e3... and then it's easy to see which one is missing from the output.

The `expected-warning`, `expected-note`, `CHECK`, and `RUN` comments are actually meaningful to Clang's testing harness.

`expected-note@-2{{text}}` means "expect to see a note emitted 2 lines up from here, containing at least `text`."

Step 3: Add the code

First, I found the code that implemented copy elision. This was easy because Clang has really good Doxygen-style comments. Here is the actual function header cut-and-pasted from `lib/Sema/SemaStmt.cpp`:

```
/// \brief Perform the initialization of a potentially-movable value, which
/// is the result of return value.
///
/// This routine implements C++14 [class.copy]p32, which attempts to treat
/// returned lvalues as rvalues in certain cases (to prefer move construction),
/// then falls back to treating them as lvalues if that failed.
ExprResult
Sema::PerformMoveOrCopyInitialization(const InitializedEntity &Entity,
                                      const VarDecl *NRVOCandidate,
                                      QualType ResultType,
                                      Expr *Value,
                                      bool AllowNRVO);
```

Step 3: Add the code

Here is what the code looked like before my patch (modulo some software engineering):

```
ExprResult Sema::PerformMoveOrCopyInitialization(const InitializedEntity &Entity,
    const VarDecl *NRVOCandidate, QualType ResultType, Expr *Value, bool AllowNRVO) {
  ExprResult Res = ExprError();

  if (AllowNRVO) {
    if (!NRVOCandidate)
      NRVOCandidate = getCopyElisionCandidate(ResultType, Value, CES_Default);
    if (NRVOCandidate)
      AttemptMoveInitialization(*this, Entity, NRVOCandidate, ResultType, Value, false, Res);
  }

  // Either we didn't meet the criteria for treating an lvalue as an rvalue,
  // above, or overload resolution failed. Either way, we need to try
  // (again) now with the return value expression as written.
  if (Res.isInvalid())
    Res = PerformCopyInitialization(Entity, SourceLocation(), Value);

  return Res;
}
```

Step 3: Add the code

And after my patch:

```
if (AllowNRVO) {
  if (!NRVOCandidate)
    NRVOCandidate = getCopyElisionCandidate(ResultType, Value, CES_Default);
  if (NRVOCandidate)
    AttemptMoveInitialization(*this, Entity, NRVOCandidate, ResultType, Value, false, Res);
  if (Res.isInvalid()) {
    auto *FakeCandidate = getCopyElisionCandidate(QualType(), Value, CES_AsIfByStdMove);
    if (FakeCandidate) {
      ExprResult FakeRes = ExprError();
      AttemptMoveInitialization(*this, Entity, FakeCandidate, ResultType, Value, true, FakeRes);
      if (!FakeRes.isInvalid()) {
        bool IsThrow = (Entity.getKind() == InitializedEntity::EK_Exception);
        Diag(Value->getExprLoc(), diag::warn_return_std_move)
            << Value->getSourceRange()
            << FakeCandidate->getDeclName() << IsThrow;
      }
    }
  }
}

// Either we didn't meet the criteria for treating an lvalue as an rvalue, ...
if (Res.isInvalid())
  Res = PerformCopyInitialization(Entity, SourceLocation(), Value);
return Res;
```

If no acceptable constructor was found, we go on and look for an *unacceptable* constructor or conversion operator, via super lenient rules.

We won't *use* `FakeRes`, but if we find that it exists, we can warn the user.

Recall the format of our diagnostic: "`local variable %0 will be copied despite being %select{returned|thrown}1 by name`". Here are our `%0` and `%1`.

Step 3b: Add more code

Remember that we were going to have “fixits.” How do I generate a fixit note?

```
bool IsThrow = (Entity.getKind() == InitializedEntity::EK_Exception);
Diag(Value->getExprLoc(), diag::warn_return_std_move)
    << FakeCandidate->getDeclName() << IsThrow;
```

Step 3b: Add more code

Remember that we were going to have “fixits.” How do I generate a fixit note?

```
    bool IsThrow = (Entity.getKind() == InitializedEntity::EK_Exception);
    Diag(Value->getExprLoc(), diag::warn_return_std_move)
        << FakeCandidate->getDeclName() << IsThrow;

    SmallString<32> Str;
    Str += "std::move(";
    Str += FakeCandidate->getDeclName().getAsString();
    Str += ")";
    Diag(Value->getExprLoc(), diag::note_add_std_move)
        << FixItHint::CreateReplacement(Value->getSourceRange(), Str);
```

Done! Easy!

Step 3c: Add more code

- Don't suggest `std::move` if the "copy constructor" that was found is trivial, because the code is already optimal.

```
String test(int i) {
    double d = 3.14;
    return d;           // std::move(d) would be silly
}
```

- Don't suggest `std::move` if the variable being returned is of *lvalue reference* type, because that suggestion would be wrong.

```
String test(String *p) {
    String &x = *p;
    return x;        // std::move(x) would be so very wrong
}
```

Step 4: Ship it!

https://reviews.llvm.org/D43322

As of May 2018,
`-Wreturn-std-move` is present in Clang trunk
and enabled by `-Wall`.

In San Diego, David Stone will present P1155R0 “More implicit moves” and P0527R1 “Implicitly move from rvalue references in return statements.”

Step 5: Write a paper!

In San Diego, David Stone will present two papers:

- P1155R0 “More implicit moves”
- P0527R1 “Implicitly move from rvalue references in return statements”

The following slides show some examples from P1155.

Example 1

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

Widget six(Widget w) {
    return w;
}

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

Copy elision?	Implicit move?	Plain old copy?

Example 1

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

Widget six(Widget w) {
    return w;
}

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

P1155 proposes to make `seven` an implicit move.

Copy elision?	Implicit move?	Plain old copy?
Technically permitted	✓	
	✓	
	Clang, MSVC, Intel	✓

Example 2

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

From eight() {
    Widget w;
    return w;
}
Widget nine() {
    To t;
    return t;
}
```

Copy elision?	Implicit move?	Plain old copy?

Example 2

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

From eight() {
    Widget w;
    return w;
}
Widget nine() {
    To t;
    return t;
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
		✓

P1155 proposes to make both of these do implicit move.

Example 3

```
struct Fish {
    From(Widget const&);
    From(Widget&&);
};
struct Fowl {
    Fowl(Widget);
};

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

Copy elision?	Implicit move?	Plain old copy?

Example 3

```
struct Fish {
    From(Widget const&);
    From(Widget&&);
};
struct Fowl {
    Fowl(Widget);
};

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

Copy elision?	Implicit move?	Plain old copy?
	✓	
	GCC	✓

P1155 proposes to make both of these do implicit move.

Example 4

```
struct Base {
    Base(Base const&);
    Base(Base&&);
};
struct Derived : Base {};

unique_ptr<Base> twelve() {
    unique_ptr<Derived> p;
    return p;
}

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

Copy elision?	Implicit move?	Plain old copy?

Example 4

```
struct Base {
    Base(Base const&);
    Base(Base&&);
};
struct Derived : Base {};

unique_ptr<Base> twelve() {
    unique_ptr<Derived> p;
    return p;
}

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

P1155 proposes to make both of these do implicit move.

Copy elision?	Implicit move?	Plain old copy?
	✓	
	GCC, Intel	✓

Example 5

```
Widget fourteen(Widget&& w) {
    return w;
}

Widget fifteen(Widget& w) {
    Widget&& x = std::move(w);
    return x;
}
```

Copy elision?	Implicit move?	Plain old copy?

Example 5

```
Widget fourteen(Widget&& w) {
    return w;
}

Widget fifteen(Widget& w) {
    Widget&& x = std::move(w);
    return x;
}
```

Copy elision?	Implicit move?	Plain old copy?
		✓
		✓

P0527 proposes to make both of these do implicit move, instead of copy.

Example 6

```
Widget sixteen(Widget w) {
    w += 1;
    return w;
}

Widget seventeen(Widget w) {
    return w += 1;
}
```

Copy elision?	Implicit move?	Plain old copy?

Example 6

```
Widget sixteen(Widget w) {
    w += 1;
    return w;
}

Widget seventeen(Widget w) {
    return w += 1;
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
		✓

There's no proposal for this one yet, as far as I know.

Questions?