The Complete Guide to `return x;`

Arthur O'Dwyer

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

The Complete Guide to `return x;`

I also do C++ training!

`arthur.j.odwyer@gmail.com`

Arthur O’Dwyer
2021-05-04

Outline

- The “return slot”; NRVO; C++17 “deferred materialization” [4–23]
- C++11 implicit move [24–29]. Question break.
- Problems in C++11; solutions in C++20 [30–46]. Question break.
- The `reference_wrapper` saga; pretty tables of vendor divergence [47–55]
- Quick sidebar on coroutines and related topics [56–65]. Question break.
- P2266 proposed for C++23 [66–79]. Questions!

Hey look!
Slide numbers!

x86-64 calling convention

```
int f()
{
    int i = 42;
    return i;
}

int test()
{
    int j = f();
    return j + 1;
}
```

```
_Z1fv:
  movl $42, -4(%rsp)
  movl -4(%rsp), %eax
  retq

_Z4testv:
  callq _Z1fv
  addl $1, %eax
  retq
```

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

x86-64 calling convention

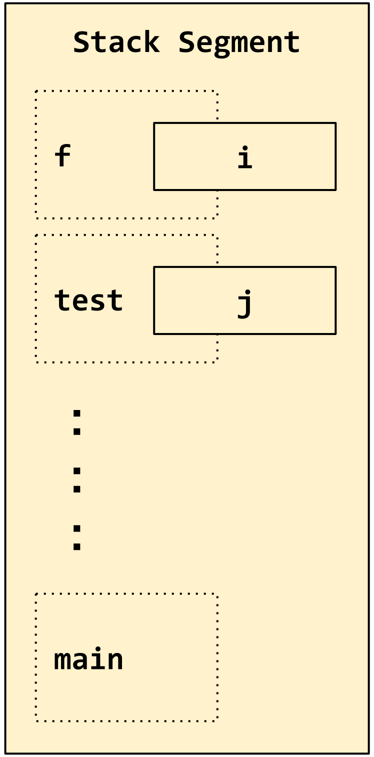
```
int f() {
    int i = 42;
    printf("%p\n", &i);
    return i;
}

int test() {
    int j = f();
    printf("%p\n", &j);
    return j + 1;
}
```

prints “0x9ff00020”

prints “0x9ff00040”

Since `f` and `test` each have their own stack frame, `i` and `j` naturally are different variables.

`j` is initialized with a **copy** of `i` — C++ loves *copy semantics*.



x86-64 calling convention

```
struct S { int m; };

S f() {
    S i = S{42};
    printf("%p\n", &i);
    return i;
}
```

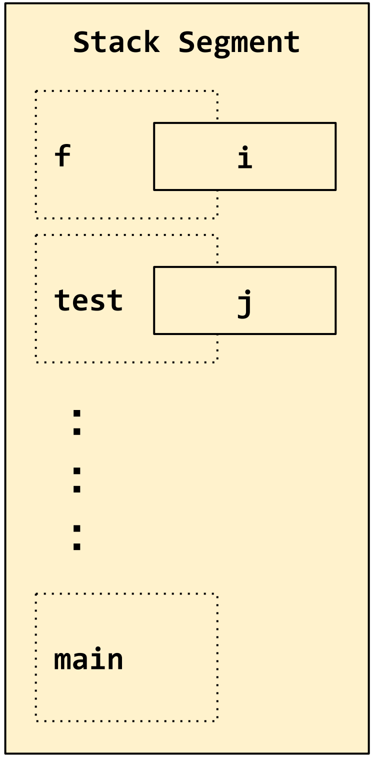
prints “0x9ff00020”

```
S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

prints “0x9ff00040”

Even for class types, C++ does “return by copy.”

The return value is ***still*** passed in a machine register when possible.



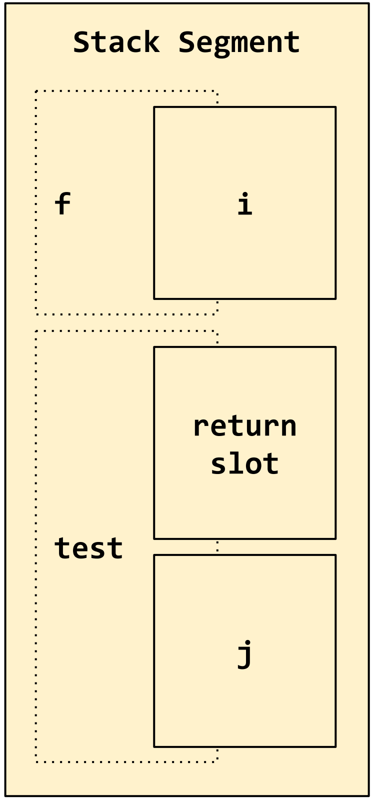
x86-64 calling convention

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

But what about when S is too big to fit in a register?

Then x86-64 says that the caller should pass an extra parameter, pointing to space ***in the caller's own stack frame*** big enough to hold the result.

This is the "return slot."

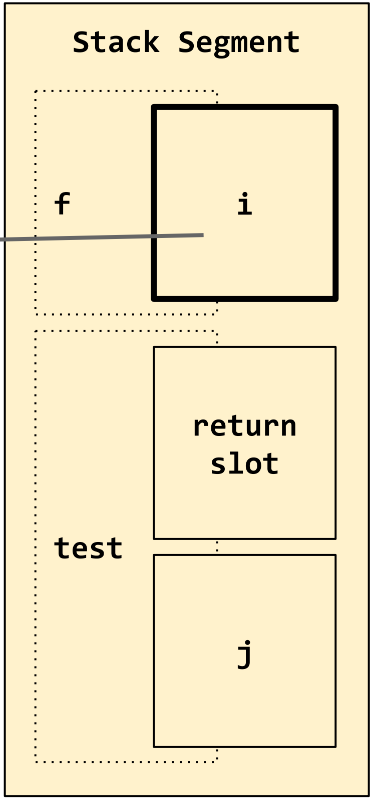


x86-64 calling convention

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

The following explanation is the baseline C++98 explanation, with no optimizations or tricks.

Don't worry, the tricks are coming.

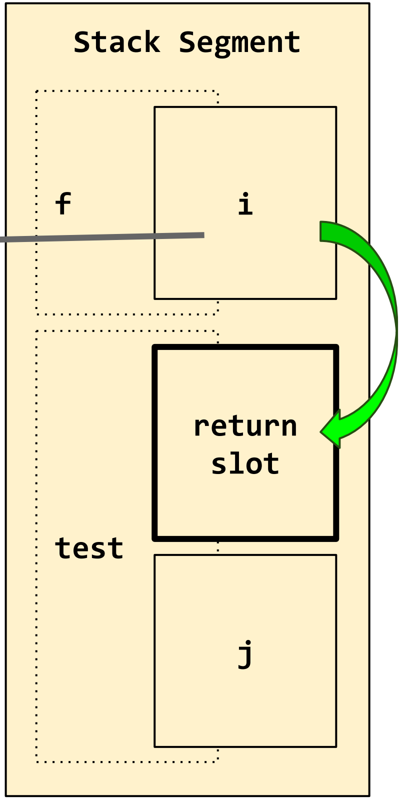


x86-64 calling convention

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

`f` knows the return slot's address because `test` passed that address to `f` as a hidden parameter (in register `%rdi`).

At the return statement, `i`'s value is copied into the return slot.

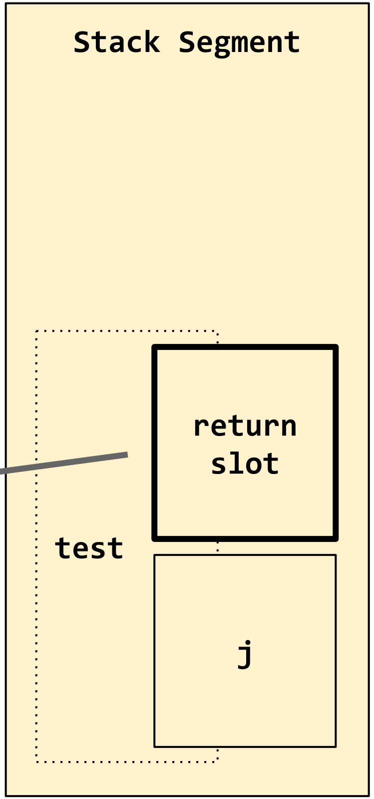


x86-64 calling convention

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

Now `f` is done and `i` is gone.

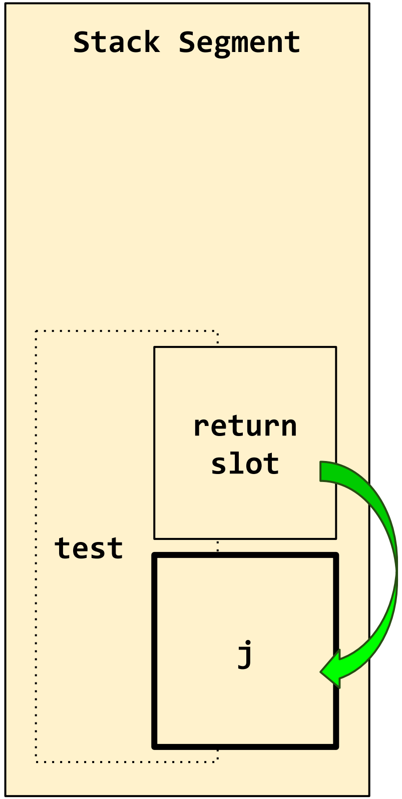
Think of the value of `f()` as the value “in the return slot.”



x86-64 calling convention

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

Finally, the value in the return slot is used to copy-initialize `j`.



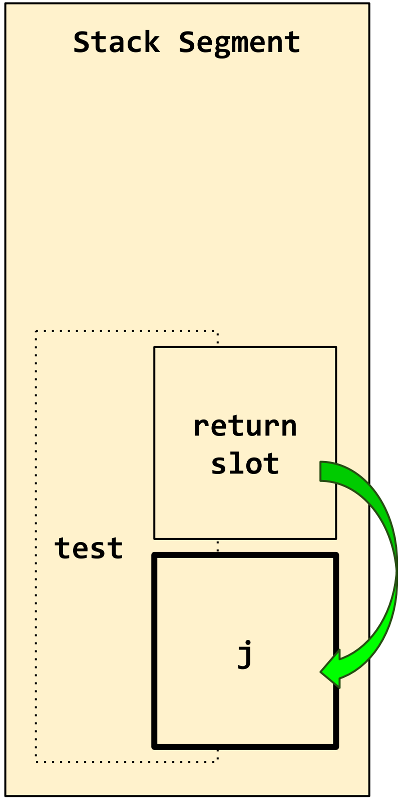
Again! Faster!

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

This step is slow.

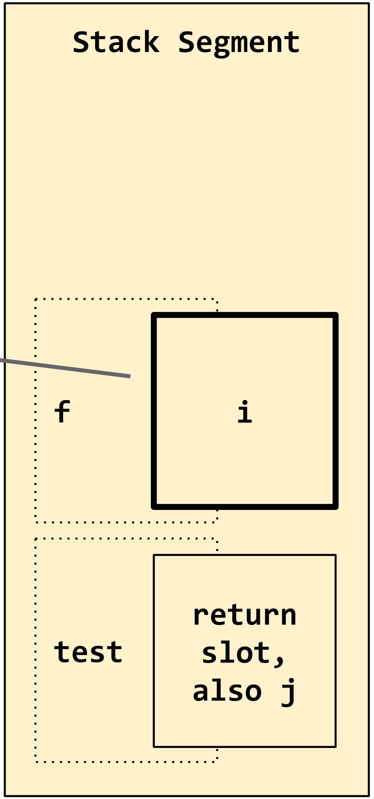
`test` can do better.

Since `test` controls the allocations of both the return slot and `j`, and `test` knows that the return slot will be used to copy-initialize `j`, `test` can allocate them both at the same memory address and avoid having to do the copy!



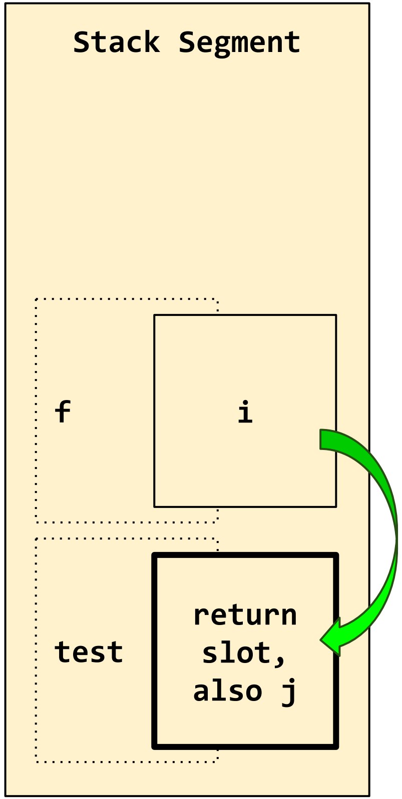
Again! Faster!

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```



Again! Faster!

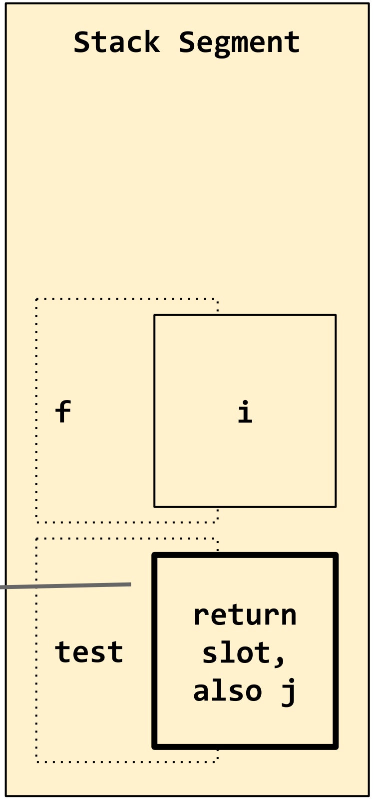
```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```



Again! Faster!

```
struct S { int m[3]; };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

And we're done!
Notice that the optimizer can do this optimization all by itself, **under the as-if rule**, because the "copying" of S has no user-visible side effects.

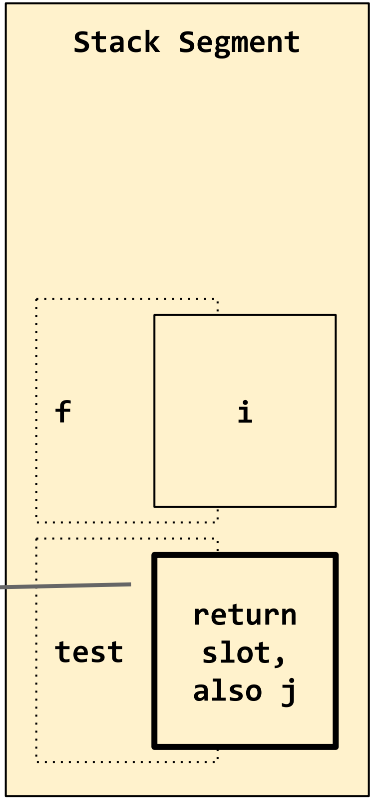


C++98 “copy elision”

```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

If we give the “copy” visible side effects, guess what? We can still do the optimization! The C++98 standard permitted us to ***elide*** even a visible constructor call when initializing an object with a temporary of the same type.

But wait, C++17 made it even better!...



C++17 “deferred prvalue materialization”

```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

C++17 changed the high-level formal semantics of a prvalue expression like “`f()`”.

In C++14, `f()` eagerly evaluated into a temporary object that had to be ***moved*** into `j`. Copy-elision was permitted by a special case.

In C++17, a prvalue is more like a ***recipe*** for initializing an object, known as the expression’s “result object.”

Here, that result object ultimately turns out to be `j` itself.

The formal semantics changed.
The machine code did not!

Again! Faster!

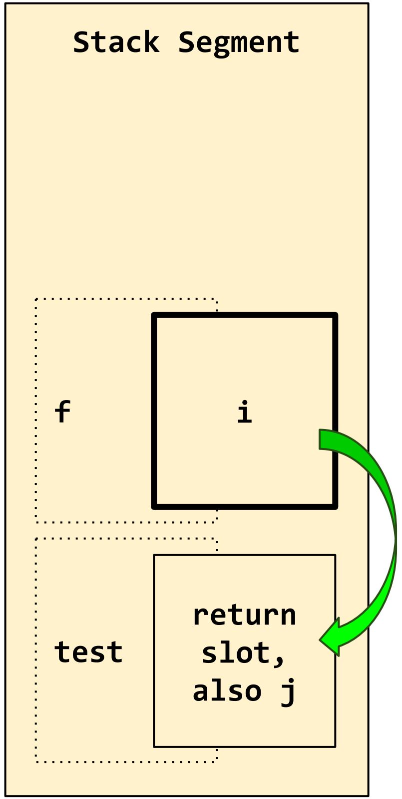
```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

This step is slow.

`f` can do better.

Since `f` controls the allocation of `i`, and knows that it will be used to initialize the return slot, `f` can allocate `i` ***in*** the return slot, and avoid having to do the copy!

Let's run through that...

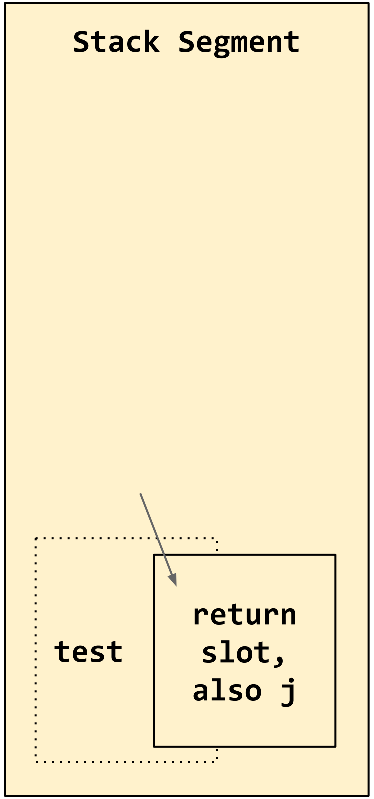


Named Return Value Optimization

```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

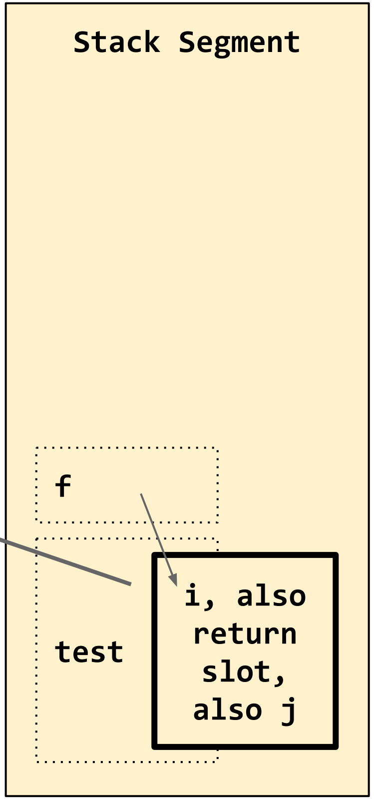
`test` allocates its stack frame as usual, and passes `f` a pointer to the return slot, in which `f` will construct its result.

(...in which the *result object* of the prvalue `f()` will be materialized.)



Named Return Value Optimization

```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```



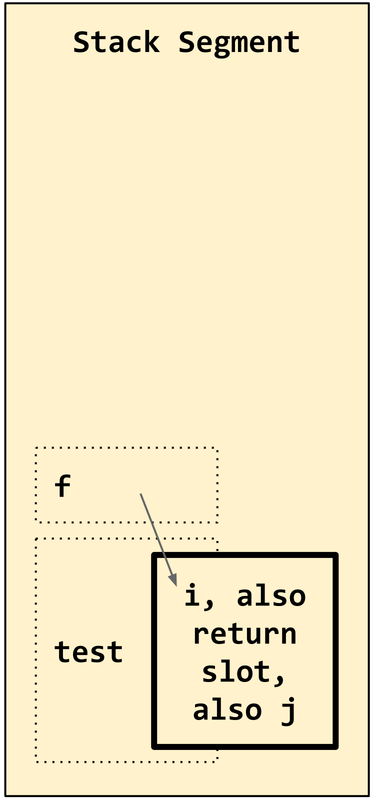
Named Return Value Optimization

```
struct S { int m[3]; S(S&&); };

S f() {
    S i = S{{1,3,5}};
    printf("%p\n", &i);
    return i;
}

S test() {
    S j = f();
    printf("%p\n", &j);
    return j;
}
```

`i` is already located in the return slot, so this `return` statement corresponds to zero machine code.

This optimization is done automatically under the as-if rule, but even when the constructor call would be visible, C++98 permits the compiler to ***elide*** it as a special case.

Note that `i` is not a prvalue, so this was ***not*** affected by C++17's "deferred materialization" business.



Conditions for NRVO to kick in

Many things have to go right for NRVO to happen.

- There must ***be*** a return slot. Trivial types can just be returned in registers
  - But types with non-trivial SMFs will always be returned via return slot
- The allocation of “return variable” `i` must be under `f`’s control
  - Otherwise `f` can’t allocate `i` into the return slot!
- `i` must have the exact same type (modulo cv-qualification) as `f`’s return slot
  - Otherwise `i` won’t ***fit*** into the return slot!
- One mental — not physical — caveat: The `return`’s operand must be exactly a (possibly parenthesized) *id-expression*, such as `i`. Nothing more complicated.
  - Otherwise things could get very confusing for the human programmer!

Examples of NRVO not happening

```
struct Trivial { int m; };
struct S { S(); ~S(); };
struct D : public S { int n; };

Trivial f() { Trivial x; return x; } // no return slot

S x;
S g1() { return x; } // g doesn’t control allocation of x

S g2() { static S x; return x; } // same deal

S g3(S x) { return x; } // same deal (params are caller-allocated!)

S h() { D x; return x; } // D is too big for the return slot
```

Introducing move semantics

- C++11 added move constructors and move-only types, such as `unique_ptr`. This was a problem for NRVO!

```
unique_ptr<T> f() {
    unique_ptr<T> x = ~~~;
    return x;
}
```

- Here x is an lvalue, not an rvalue.
- `unique_ptr` is move-only; you ***can't*** construct one from an lvalue.
- To make it an rvalue, we have to say *`return std::move(x)`* instead.
- But NRVO works only on simple *`return x`* !

Introducing *implicit move*

C++11's solution: When we see `return x`, although `x` ***is*** an lvalue, we will do a preliminary overload resolution looking for ***move constructors***. If we find one, the `return` is well-formed. If not, go try the C++03 rules.

> ... 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 ...
>
> — N3337, [class.copy]/32

```
unique_ptr<T> f() {
    unique_ptr<T> x = ~~~;
    return x;
}
```

Overload resolution finds `unique_ptr(unique_ptr&&)`, so this is well-formed. Then, the "copy elision" special case kicks in and elides the physical call.

Introducing *implicit move*

> ... 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 ...
>
> — N3337, [class.copy]/32

```
struct auto_ptr {
    auto_ptr(auto_ptr&);  // only from non-const lvalues
};
auto_ptr f() { auto_ptr x; return x; }
```

Since `auto_ptr` isn't constructible from an rvalue `auto_ptr`, we fall back to considering x as an lvalue (just like in C++03), and everything works.

Introducing *implicit move*

> ... 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 ...
>
> — N3337, [class.copy]/32

```
struct AutoSharePtr {
    AutoSharePtr(AutoSharePtr&);  // pre-'11 "fast pilfer"
    AutoSharePtr(const AutoSharePtr&);  // or copy, if we must
};
AutoSharePtr f() { AutoSharePtr x; return x; }
```

In C++98, `return x` would use `AutoSharePtr`'s fast "move" constructor, because x is a non-const lvalue. Resolving first "as if x were an rvalue" will successfully find the ordinary (slow) copy constructor. We don't want to silently switch to calling the slow constructor!

“Return by converting move ctor”

- CWG issue 1579 broadened the rule slightly so that *implicit move* would apply even in cases where *copy elision* was never on the table.

```
unique_ptr<Base> g3(unique_ptr<Base> x) {
    return x;  // OK, implicit move!
}

unique_ptr<Base> h2() {
    unique_ptr<Derived> x = ~~~;
    return x;  // OK, implicit move!
}
```

C++11 enabled implicit move for function parameters of the proper type.

CWG1579 further enabled it for ***all*** local objects of automatic storage duration... ***regardless of type!***

In `h2`, the first overload resolution finds `unique_ptr(unique_ptr<U>&&) [with U=Derived]`, so this `return x` is well-formed.

Now we’re caught up to C++11, which is to say, C++17.

Questions so far?

Holes in the CWG1579 model

```
unique_ptr<Base> h2() {
    unique_ptr<Derived> x = ~~~;
    return x;  // OK, implicit move
}
```

Copy elision is not on the table.
The first overload resolution finds `unique_ptr(unique_ptr<Derived>&&)`, whose argument ***is*** an rvalue ref to x’s type.

In C++17, the above code works fine... but the following did ***not***.

```
Base h3() {
    Derived x = ~~~;
    return x;  // Ugh, copy! 🤮
}
```

Copy elision is not on the table.
The first overload resolution finds `Base(Base&&)`, whose argument ***is not*** an rvalue ref to x’s type.

This ***is*** well-formed; it simply initializes the result object via `Base(const Base&)` instead of `Base(Base&&)`.

Holes in the CWG1579 model

> ... 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 ...
>
> — N3337, [class.copy]/32

If the selected constructor takes a parameter of type "rvalue reference to one of my bases," then we'll ignore that constructor and implicit move will fail.

```
struct Base { Base(Base&&); Base(const Base&); };
struct Derived : Base {};
Base f() { Derived x; return x; }  // C++17 calls Base(const Base&)
```

Holes in the CWG1579 model

> ... 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 ...
>
> — N3337, [class.copy]/32

If the selected constructor takes a parameter of *exactly* x's type, then we'll ignore that constructor: implicit move will fail.

```
struct Source { Source(Source&&); Source(const Source&); };
struct Sink { Sink(Source); Sink(unique_ptr<int>); };
Sink f() { Source x; return x; }  // C++17 calls Source(const Source&),
                                  // then Sink(Source)
Sink g() { unique_ptr<int> p; return p; }  // C++17: ill-formed
```

Holes in the CWG1579 model

> ... 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 ...
>
> — N3337, [class.copy]/32

If overload resolution succeeds by finding a non-constructor, implicit move will fail!

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

To f() { From x; return x; }  // C++17 calls From::operator To() const&
```

Seems contrived, but we really hit this in github.com/WG21-SG14/SG14 #125.
Conversion operators not a perfect substitute for converting constructors: who knew?!

P1155 “More implicit move”

So I put in a paper for C++20. It was nothing but deletions. It was adopted.

> ... 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 ...

```
Base f() { Derived x; return x; }  // C++20 calls Base(Base&&)

Sink f() { Source x; return x; }  // C++20 calls Source(Source&&),
                                  // then Sink(Source)

To f() { From x; return x; }  // C++20 calls From::operator To() &&
```

P1155 “More implicit move”

P1155 actually deleted a tiny bit more, in text we haven’t seen yet!

See, implicit move doesn’t ***just*** apply to `return`. It applies to any situation where the compiler can tell (purely lexically) that this is the last use of the object before leaving the function.

That includes `throw x`, and in C++20, `co_return x` as well.

> If the *expression* in a `return` or `co_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*, or
>
> if the operand of a *throw-expression* 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...

Implementation divergence galore

By the way, when I say “C++17 says...,” I’m talking about the paper standard.
Actual vendors already implemented random subsets of P1155,
either by user demand or simply because the C++17 rules were so *weird*.

C++17 conformance as of June 2019	GCC	Clang	MSVC	Intel ICC
`void f(T x) { throw x; }`	copy	move	move	move
`Base f() { Derived x; return x; }`	move	move	copy	copy
`To f() { From x; return x; }`	copy	copy	copy	copy
`Sink f() { Source x; return x; }`	move	copy	copy	copy

Implementation divergence galore

Vendors' conformance to the C++17 rules drifts randomly over time...

C++17 conformance as of May 2021	GCC 11	Clang 12	VS 16.9	ICC 2021.1
`void f(T x) { throw x; }`	copy	move	move	move
`Base f() { Derived x; return x; }`	copy	copy	move	move
`To f() { From x; return x; }`	move	copy	copy	copy
`Sink f() { Source x; return x; }`	move	copy	move	copy

Implementation divergence galore

But I really think P1155 helped vendors converge...
at least if you look only at their C++20 mode!
The C++20 rules are *just simpler*.

Yellow indicates "fixed in trunk"

C++20 conformance as of May 2021	GCC 11	Clang 12	VS 16.9	ICC 2021.1
`void f(T x) { throw x; }`	move	move	move	move
`Base f() { Derived x; return x; }`	move	copy	move	move
`To f() { From x; return x; }`	move	copy	copy	copy
`Sink f() { Source x; return x; }`	move	copy	move	copy

C++20 made another big change!

David Stone's paper P0527 "Implicitly move from rvalue references in return statements" factored out the notion of an *implicitly movable entity*.

> An *implicitly movable entity* is a variable of automatic storage duration and non-volatile object type.
>
> If the *expression* in a `return` or `co_return` statement is a (possibly parenthesized) *id-expression* that names an ~~object with automatic storage duration~~ implicitly movable entity declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, or
>
> if the operand of a *throw-expression* is ~~the name of a non-volatile automatic object (other than a catch-clause parameter)~~ a (possibly parenthesized) *id-expression* that names an implicitly movable entity whose scope does not extend beyond ...

C++20 made another big change!

And then dropped this bombshell:

An *implicitly movable entity* is a variable of automatic storage duration that is **either** a non-volatile object **or an rvalue reference** to a non-volatile object type.

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

if the operand of a *throw-expression* is ~~the name of a non-volatile automatic object (other than a catch-clause parameter)~~ a (possibly parenthesized) *id-expression* that names an implicitly movable entity whose scope does not extend beyond ...

Rvalue refs are now implicit-movable

Recall that in C++, *anything with a name* is an lvalue.

```
std::string setName(std::string&& rr)
{
    name_ = rr;   // copies from the-string-referred-to-by-rr...
    id_ = rr;     // ...which is good, because we might use rr again right here
    return rr;    // rr is an lvalue here, too (in C++17)
}
```

But “we might use `s` again” doesn’t apply in a `return` or `throw`!

(This is the same reasoning that permitted “implicit move” in the first place.)

In C++20, `return rr` triggers the implicit-move rules.

“Perfect backwarding” (version 1)

```
template<class T>
auto f(T t) {
    return t.foo();
}

template<class T>
auto g(T t) {
    decltype(auto) x = t.foo();
    return x;
}

struct A { A foo(); };
struct B { B& foo(); };
struct C { C&& foo(); };
```

`f<A>` initializes its result object with `t.foo()`’s prvalue result of type `A`.

`g<A>` defines `x` of type `A`, then may apply NRVO (and if not, then implicit move kicks in).

`f<B>` copy-constructs from `t.foo()`’s `B&` result into its result object.

`g<B>` defines `x` of type `B&`, then copy-constructs from `x` into its result object.

`f<C>` move-constructs from `t.foo()`’s `C&&` result into its result object.

`g<C>` defines `x` of type `C&&`, then (in C++20) move-constructs from `x` into its result object.
In C++17 it would have copy-constructed!

But there’s a problem even in C++20

```
template<class T>
decltype(auto) f(T t) {
    return t.foo();
}

template<class T>
decltype(auto) g(T t) {
    decltype(auto) x = t.foo();
    return x;
}

struct A { A foo(); };
struct B { B& foo(); };
struct C { C&& foo(); };
```

`f<A>` initializes its result object with `t.foo()`’s prvalue result of type `A`.

`g<A>` defines `x` of type `A`, then may apply NRVO (and if not, then implicit move kicks in).

`f<B>` returns a `B&` bound to the-referent-of `t.foo()`’s `B&` result.

`g<B>` defines `x` of type `B&`, then returns a `B&` bound to the-referent-of `x` (which is exactly what we want).

`f<C>` returns a `C&&` bound to the-referent-of `t.foo()`’s `C&&` result.

`g<C>` defines `x` of type `C&&`, then tries to bind a `C&&` to the-referent-of-x... but `x` is an lvalue.
This `return` is ill-formed, even in C++20.

Implicit move applies only to objects

The reason `g<C>` couldn't bind an rvalue reference to `x` was that `x` was not "implicit moved." Implicit move applies only to functions that return objects!

> In the following copy-initialization contexts, a move operation is first considered before attempting a copy operation:
>
> - If the operand of a `return` or `co_return` statement is a (possibly parenthesized) *id-expression* that names an implicitly movable entity declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, or
> - if the operand of a *throw-expression* is a (possibly parenthesized) *id-expression* that names an implicitly movable entity that belongs to a scope that does not contain the *compound-statement* of the innermost *try-block* ...,
>
> overload resolution to select the constructor for the copy or the `return_value` overload to call is first performed as if the expression or operand were an rvalue...

Binding a reference is not a "copy-initialization context."

Implicit move applies only to objects

So in C++20, we have this unfortunate situation.

```
MoveOnly
one(MoveOnly&& rr)
{
    return rr;  // OK, move-constructs from rr (in C++20)
}

MoveOnly&&
two(MoveOnly&& rr)
{
    return rr;  // ill-formed, rr is an lvalue
}
```

Now we're caught up to C++20.

Questions so far?

The dangling `reference_wrapper`

```
reference_wrapper<int> f() {
    int x = 42;
    return x;
}

template<class T>
struct reference_wrapper {
    reference_wrapper(T&);
};
```

Looks simple, right?

In C++98, of course it compiled, and returned a dangling `reference_wrapper`.

The dangling `reference_wrapper`

```
reference_wrapper<int> f() {
    int x = 42;
    return x;
}

template<class T>
struct reference_wrapper {
    reference_wrapper(T&);
    reference_wrapper(T&&) = delete;
};
```

Looks simple, right?

In C++98, of course it compiled, and returned a dangling `reference_wrapper`.

In C++11 prior to CWG1579, ditto.

The dangling `reference_wrapper`

```
reference_wrapper<int> f() {
    int x = 42;
    return x;
}

template<class T>
struct reference_wrapper {
    reference_wrapper(T&);
    reference_wrapper(T&&) = delete;
};
```

Looks simple, right?

In C++98, of course it compiled, and returned a dangling `reference_wrapper`.

In C++11 prior to CWG1579, ditto.

After CWG1579, `x` is a candidate for implicit move. The first overload resolution successfully finds the deleted constructor: `return x` is **ill-formed.**

The dangling `reference_wrapper`

```
template<class T>
struct reference_wrapper {
    reference_wrapper(T&);
    reference_wrapper(T&&) = delete;
};
```

The problem with C++11 `reference_wrapper` was that deleted functions are still visible to overload resolution. We don't want that! We want it to SFINAE out of the way properly.

```
void g(reference_wrapper<unique_ptr<Derived>>);  // bind to lvalue
void g(unique_ptr<Base>);                          // bind to rvalue

int main() {
    g(make_unique<Derived>());  // oops, this is ambiguous! LWG issue 2993
}
```

The dangling `reference_wrapper`

```
reference_wrapper<int> f() {
    int x = 42;
    return x;
}

template<class T>
struct reference_wrapper {
    template<class U>
      requires VeryComplexTest<U>
    reference_wrapper(U&&);
};
```

Looks simple, right?

In C++98, of course it compiled, and returned a dangling `reference_wrapper`.

In C++11 prior to CWG1579, ditto.

After CWG1579, `x` is a candidate for implicit move. The first overload resolution successfully finds the deleted constructor: `return x` is ill-formed.

After LWG2993, the first overload resolution finds no candidates, so we do the second resolution treating `x` as an lvalue: it returns a dangling `reference_wrapper`.

Implementation divergence

Deleted functions are still visible to overload resolution. But if our first (rvalue) overload resolution finds a deleted function, should that perhaps count as “failure”?

> If the first overload resolution fails or was not performed, overload resolution is performed again, considering the expression as an lvalue.

If the best match is deleted, is that a “failure”?

```
struct RefWrap { RefWrap(T&); RefWrap(T&&) = delete; };
RefWrap f() { T x; return x; }  // ill-formed since CWG1579 (C++11)
```

If overload resolution is ambiguous, is that a “failure”?

```
struct Left {}; struct Right {}; struct Both: Left, Right {};
struct Ambig { Ambig(Left&&); Ambig(Right&&); Ambig(Both&); };
Ambig f() { Both x; return x; }  // ill-formed since P1155 (C++20)
```

Implementation divergence

And, as noted, implicit move applies only to objects.
So P1155's gains are distributed inequitably:

```
using IntR = std::reference_wrapper<int>;
struct Larry { operator IntR() const&; operator IntR() &&; };
struct Curly { operator int&() const&; operator int&() &&; };
struct Shemp { operator int*() const&; operator int*() &&; };

IntR f1() { Larry x; return x; }  // C++20:    operator IntR() &&
int& f2() { Curly x; return x; }  // Surprise! operator int&() const&
int* f3() { Shemp x; return x; }  // Surprise! operator int*() const&
```

Since binding a reference is not a "copy-initialization," the implicit-move rules from C++98 through C++20 never kick in for `Curly`.

They don't kick in for `Shemp` either, because `int*` is not a class type.

Implementation divergence

C++20 conformance as of May 2021	GCC 11	Clang 12	VS 16.9	ICC 2021.1
`RefWrap f() { T x; return x; }`	ill	ill	ill	ill
`Ambig f() { Both x; return x; }`	well	well	well	ill
`int& f() { Shemp x; return x; }`	copy	copy	copy	copy

My interpretation of these results:
Everyone knows about `RefWrap` because of LWG 2993, but `Ambig` hasn't had its breakout moment yet.
The difference between `Larry` and `Shemp` is new in C++20, so VS and ICC haven't had a chance to be confused by it yet.

Hm. Implicit move is *still* confusing

- Doing two overload resolutions is confusing for vendors.
  - What does it mean for the first one to “fail”?
- Restricting implicit move to class-type copy-initialization contexts is confusing for users.
  - Surprising contrast between `reference_wrapper<int>` and `int&`
- Speaking of `reference_wrapper`... why is this even well-formed?

```
reference_wrapper<int> f() { int x; return x; }
```

https://godbolt.org/z/vfTa5EaYW

Quick sidebar on C++20 coroutines

C++20 expanded implicit move to work on `co_return` as well as `return`.
By my reading, C++20 doesn't limit the `co_return` case to class types! So technically,

```
struct Curly { operator int&() const&; operator int&() &&; };

template<class T>
struct task {
    struct promise_type {
        void return_value(const T&);
        void return_value(T&&);
    };
};

task<int> f1() { int x; co_return x; }    // C++20: return_value(int&&)
task<int> f2() { Curly x; co_return x; }  // C++20: operator int&() &&
```

https://godbolt.org/z/vfTa5EaYW

Quick sidebar on C++20 coroutines

C++20 conformance as of May 2021	GCC 11	Clang 12	VS 16.9
`task<int> f1() { int x; co_return x; }`	move	copy	copy
`task<int> f2() { Curly x; co_return x; }`	move	move	move

What about co_yield?

You might expect implicit move to work here:

```
template<class T>
struct generator {
    struct promise_type {
        std::suspend_always yield_value(const T&);
        std::suspend_always yield_value(T&&);
    };
};

generator<std::string> g() {
    for (int i=0; i < 100; ++i) {
        std::string x = std::to_string(i);
        co_yield x;   // Hmm... Couldn't we move-from x here?
    }
}
```

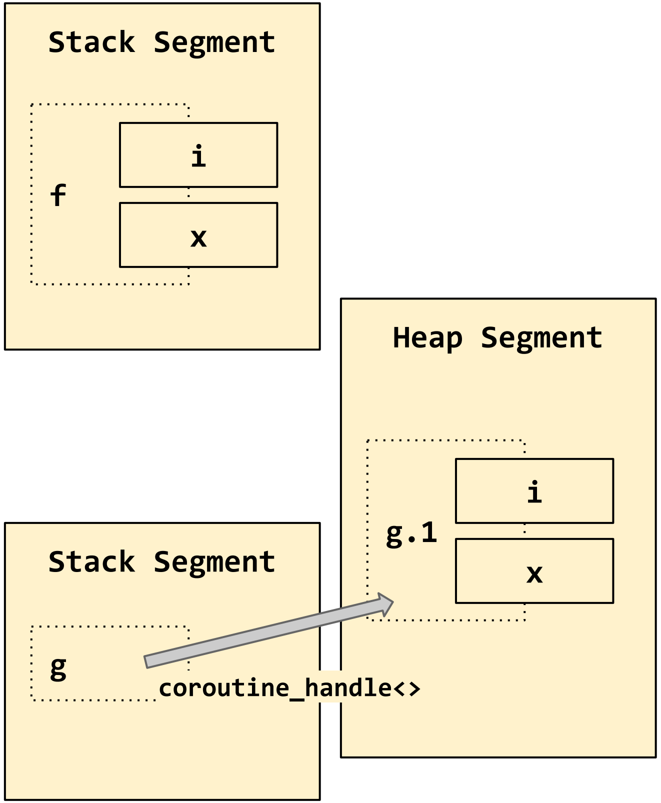
What about co_yield?

A coroutine isn't exactly like a function.

When you return from ***function*** `f`, its stack frame goes away. So you know the return is the last use of `x`.

When you `co_yield` from coroutine `g`, its "activation frame" does ***not*** go away. It goes away only when you finally `co_return`.

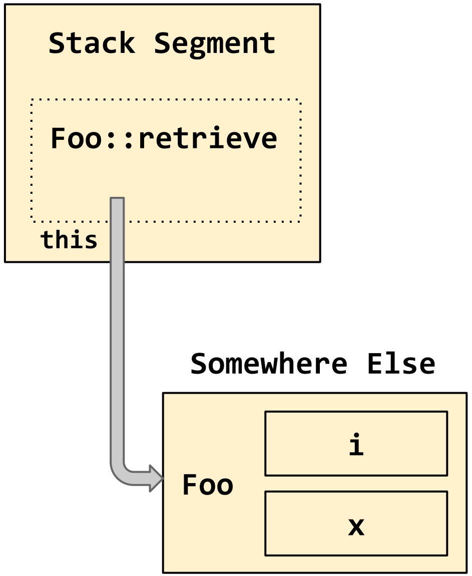
```
generator<std::string> g() {
    for (int i=0; i < 100; ++i) {
        std::string x = std::to_string(i);
        co_yield x;
        maybe_use(x).again_here();
    }
}
```



What else is like that?

```
struct Foo {
    T x;

    T retrieve_my_result() {
        return x;   // Can we move-from x here? No!
    }
};
```

- A `co_yield` is kind of like returning from a member function.
- The `Foo` instance's member data doesn't go away just because you returned from one of its member functions!
    - Not even if the member function has a suggestive name.
- `return std::move(x)` will never be entirely obsolete.



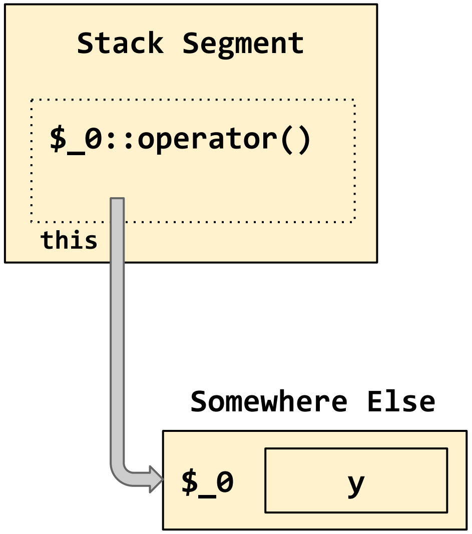
What else is like that?

```
auto lambda = [y = T()] { return y; };
```

- A lambda is just a class with member functions, in disguise.
- This is just like our very first “can’t NRVO” example:

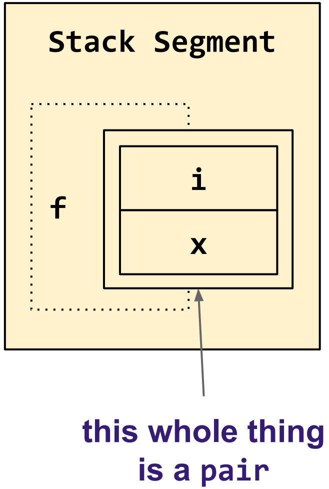
```
S x;
S f() { return x; }
```

- We don’t control allocation of x or y, so we can’t NRVO.
- We don’t control lifetime of x or y, so we can’t implicit-move.
    - Notice that “storage” and “lifetime” are not the same!



One more noteworthy case

```
std::pair<T, int> getPair();

T f() {
    auto [x, i] = getPair();
    return x;
}
```

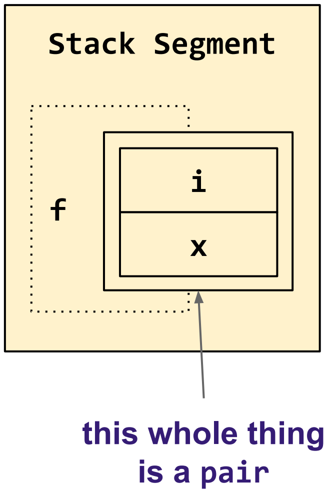


- A structured binding is a thing with data members, in disguise.
- x is syntactic sugar for (roughly) `__my_hidden_var.x`
  - So it is not a local variable, and implicit move doesn't apply!

One more noteworthy case

And then there's this:

```
std::pair<T&&, int&&> getPair();

T f() {
    auto [x, i] = getPair();
    return x;  // calls T(const T&)
}
```



- x is *still* not a variable, so C++20's new implicit move from `T&&` doesn't apply!

C++20 conformance as of May 2021	GCC 11	Clang 12	VS 16.9	ICC 2021.1
`T f() { auto [x,y] = ...; return x; }`	move	copy	copy	copy

End sidebar. Return to topic at hand.

- Doing two overload resolutions is confusing for vendors.
  - What does it mean for the first one to “fail”?
- Restricting implicit move to class-type copy-initialization contexts is confusing for users.
  - Surprising contrast between `reference_wrapper<int>` and `int&`
- Speaking of `reference_wrapper`... why is this even well-formed?

```
reference_wrapper<int> f() { int x; return x; }
```

I’m about to present a solution (P2266, likely to happen in C++23)

Questions so far?

P2266 “Simpler implicit move”

- Doing two overload resolutions is confusing
  - So, stop doing two!
  - Programmers have come to expect implicit move over the past decade
  - Programmers have caught up to move semantics and no longer write auto_ptr-style classes that depend on mutable-lvalue “copy.”
  - Remove the fallback

P2266 “Simpler implicit move”

- Restricting implicit move to class-type copy-initialization contexts is confusing for users
    - [class.copy] was never the appropriate place for the “implicit move” wording
    - It should go somewhere that applies uniformly to all types, class and primitive alike
    - [expr.prim.id.unqual] looks like a good spot

P2266 “Simpler implicit move”

We move the wording to [expr.prim.id.unqual]:

> An ***implicitly movable entity*** is a variable of automatic storage duration that is either a non-volatile object or an rvalue reference to a non-volatile object type. In the following ~~copy-initialization~~ contexts, an *id-expression* is ***move-eligible***:
>
> If the *id-expression* (possibly parenthesized) is the operand of a `return` or `co_return` statement, and names an implicitly movable entity declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, or
>
> if the *id-expression* (possibly parenthesized) is the operand of a *throw-expression*, and names an implicitly movable entity that belongs to a scope that does not contain the *compound-statement* of the innermost *lambda-expression*, *try-block*, or *function-try-block* (if any) whose *compound-statement* or *ctor-initializer* encloses the *throw-expression*.

And then we say:

> An *id-expression* is an **xvalue if it is move-eligible**; an lvalue if the entity is a function, variable, structured binding, data member, or template parameter object; and a prvalue otherwise; it is a bit-field if the identifier designates a bit-field.

P2266 “Simpler implicit move”

The wording we didn’t take up in the move, we delete!

> In the following ~~copy-initialization~~ contexts, ~~a move operation is first considered before attempting a copy operation:~~
>
> ...
>
> ~~overload resolution to select the constructor for the copy or the `return_value` overload to call is first performed as if the expression or operand were an rvalue. If the first overload resolution fails or was not performed, overload resolution is performed again, considering the expression or operand as an lvalue.~~

We simply don’t do the two overload resolutions anymore.

When you say `return x`, if `x` is move-eligible, the expression `x` ***is an rvalue.***

Otherwise, it ***is an lvalue.***

P2266’s wording applies equally to primitive types and reference bindings.
P2266’s wording doesn’t depend on the function’s return type.

The dangling `reference_wrapper`

```
reference_wrapper<int> f() {
    int x = 42;
    return x;
}

template<class T>
struct reference_wrapper {
    template<class U>
      requires ComplexTest<U>
    reference_wrapper(U&&);
};
```

In C++98, it compiled, and returned a dangling `reference_wrapper`.

In C++11 prior to CWG1579, ditto.

After CWG1579, `x` is a candidate for implicit move. The first overload resolution successfully finds the deleted constructor: `return x` is ill-formed.

After LWG2993, the first overload resolution finds no candidates, so we do the second resolution treating `x` as an lvalue: it returns a dangling `reference_wrapper`.

After P2266, `x` is an xvalue, not an lvalue, so (the only) overload resolution finds no candidates: `return x` is ill-formed. Hooray!

P2266's effects on pathological code

```
int& f() {
    int x = 42;
    return x;
}

const int& g() {
    int x = 42;
    return x;
}

int&& h() {
    int x = 42;
    return x;
}
```

In C++98 through C++20, `f` and `g` are well-formed and return dangling references.
h is ill-formed.

After P2266, `g` and h are well-formed and return dangling references.
`f` is ill-formed.

All three are dangerous and silly. P2266 merely shuffles around the ill-formedness of these dangerous functions.

P2266’s effects on pathological code

```
struct auto_ptr {
    auto_ptr(auto_ptr&);
};

auto_ptr f() {
    auto_ptr x;
    return x;
}
```

In C++98 through C++20, `f` is well-formed.

After P2266, `f` is ill-formed!

To make this code acceptable to P2266 (C++23?), you must return something that is not an *id-expression*.

```
return auto_ptr(x);  // OK

return static_cast<auto_ptr&>(x);
                     // OK
```

C++20’s effects on pathological code

```
struct AutoSharePtr {
    AutoSharePtr(AutoSharePtr&);
    AutoSharePtr(const AutoSharePtr&);
};

AutoSharePtr f() {
    AutoSharePtr x;
    return x;
}
```

In C++98 through C++17, `f` is well-formed and calls `AutoSharePtr(AutoSharePtr&)`.

In C++20, `f` is well-formed and calls `AutoSharePtr(const AutoSharePtr&)`. Because the first overload resolution treats `x` as an rvalue and finds that candidate!

```
return AutoSharePtr(x); // better
```

P2266 (C++23?) does not change the behavior of this code.

“Perfect backwarding” (version 1)

```
template<class T>
auto f(T t) {
    return t.foo();
}

template<class T>
auto g(T t) {
    decltype(auto) x = t.foo();
    return x;
}

struct A { A foo(); };
struct B { B& foo(); };
struct C { C&& foo(); };
```

`f<A>` initializes its result object with `t.foo()`’s prvalue result of type `A`.

`g<A>` defines `x` of type `A`, then may apply NRVO (and if not, then implicit move kicks in).

`f<B>` copy-constructs from `t.foo()`’s `B&` result into its result object.

`g<B>` defines `x` of type `B&`, then copy-constructs from `x` into its result object.

`f<C>` move-constructs from `t.foo()`’s `C&&` result into its result object.

`g<C>` defines `x` of type `C&&`, then (in C++20) move-constructs from `x` into its result object.
In C++17 it would have copy-constructed!

“Perfect backwarding” (version 2)

```
template<class T>
decltype(auto) f(T t) {
    return t.foo();
}

template<class T>
decltype(auto) g(T t) {
    decltype(auto) x = t.foo();
    return x;
}

struct A { A foo(); };
struct B { B& foo(); };
struct C { C&& foo(); };
```

`f<A>` initializes its result object with `t.foo()`’s prvalue result of type `A`.

`g<A>` defines `x` of type `A`, then may apply NRVO (and if not, then implicit move kicks in).

`f<B>` returns a `B&` bound to the-referent-of `t.foo()`’s `B&` result.

`g<B>` defines `x` of type `B&`, then returns a `B&` bound to the-referent-of `x`. (`x` is not move-eligible.)

`f<C>` returns a `C&&` bound to the-referent-of `t.foo()`’s `C&&` result.

`g<C>` defines `x` of type `C&&`, then returns a `C&&` bound to the-referent-of `x`. (Because `x` is move-eligible!)
In C++20 this was ill-formed. P2266 makes it work.

P2266 and `decltype(auto)`

```
decltype(auto) f() {
    T x;
    return x;
}

decltype(auto) g() {
    T x;
    return (x);
}
```

In all versions of C++, `decltype(auto)` follows the same special case as `decltype(`*`expr`*`)`.

If the thing-being-decltyped is an *id-expression* that names an entity, we use the declared type of the entity without considering the value category of the *id-expression*.

All versions of C++ make it `T f()`.

But up to C++20, it's been `T& g()`.

After P2266 (C++23?), it's `T&& g()`.

P2266 and `decltype(`*expr*`)`

Return type, ill-formed, well-formed	C++14, 17, 20	P2266 (C++23)
`auto a(T x) -> decltype(x) { return x; }`	T	T
`auto b(T x) -> decltype((x)) { return (x); }`	T&	T&
`auto c(T x) -> decltype(auto) { return x; }`	T	T
`auto d(T x) -> decltype(auto) { return (x); }`	T&	**T&&**
`auto e(T&& x) -> decltype(x) { return x; }`	T&&	T&&
`auto f(T&& x) -> decltype((x)) { return (x); }`	T&	T&
`auto g(T&& x) -> decltype(auto) { return x; }`	T&&	T&&
`auto h(T&& x) -> decltype(auto) { return (x); }`	T&	**T&&**

P2266 almost implemented in Clang

- P2266 went to EWG in March 2021
    - Generally approved, targeting C++23, on the major condition that we get some implementation experience.
- Matheus Izvekov has been implementing P2266 in Clang
    - His patch: `reviews.llvm.org/D99005`
    - It's ***extremely unlikely*** (but not *completely impossible*) that we land this in `-std=c++2b` mode before P2266 is accepted
- We need volunteers to try out this patched Clang!
    - Especially if they're C++Now sponsors with 1990s-era `ManagedPtr` types

That’s pretty much it!

Questions?

Bonus slides

Rvalue doesn't imply non-const

Returning an implicitly movable entity by name, in a copy-initialization context, causes overload resolution to treat the entity as an rvalue if possible.

This doesn't change the entity's `constness`!

```
Fruit f() {
    const Durian cd;
    return cd;
}
```

The expression `cd` is an lvalue `const Durian&`,
but when we do implicit move, we'll first look it up as if it was an rvalue `const Durian&&`.

Overload resolution finds the same copy constructor,
`Fruit(const Fruit&)`,
that it would have found in the lvalue case.

Guaranteed NRVO in C++23?

Anton Zhilin's P2025 "Guaranteed copy elision for named return objects" was briefly pronounced "tentatively ready" for C++23 and sent to CWG, but bounced back.

```
X test() {
    X a;
    if (rand()) {
        X b;
        if (rand()) return b;
    }
    if (rand()) return a;
    X c;
    return c;
}
```

```
X result = test();
// &b == &c == &result
```

Within the potential scope of b, every `return` is a `return b`. Therefore b is called a ***named return object***.
c is also a named return object.
Both b and c would be guaranteed NRVO under P2025.

Guaranteed NRVO in C++23?

Today, Clang is better than the rest at NRVO.
P2025 asks for ***everyone*** to get ***even better*** than Clang is today.

```
X test() {
    X a;
    if (rand()) {
        X b;
        if (rand()) return b;
    }
    if (rand()) return a;
    X c;
    return c;
}
```

Clang does copy elision here.
GCC/ICC do not.

Nobody does copy elision here.

Remember from slide 35: Copy elision never applies to objects that are returned in registers. “Guaranteed NRVO” means “guaranteed for non-trivial class types.”

Really.
Questions?