16.5 — Returning std::vector, and an introduction to move semantics

When we need to pass a `std::vector` to a function, we pass it by (const) reference so that we do not make an expensive copy of the array data.

Therefore, you will probably be surprised to find that it is okay to return a `std::vector` by value.

Say whaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaat?

Copy semantics

Consider the following program:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector arr1 { 1, 2, 3, 4, 5 }; // copies { 1, 2, 3, 4, 5 } into arr1
    std::vector arr2 { arr1 };          // copies arr1 into arr2

    arr1[0] = 6; // We can continue to use arr1
    arr2[0] = 7; // and we can continue to use arr2

    std::cout << arr1[0] << arr2[0] << '\n';

    return 0;
}
```

When `arr2` is initialized with `arr1`, the copy constructor of `std::vector` is called, which copies `arr1` into `arr2`.

Making a copy is the only reasonable thing to do in this case, as we need both `arr1` and `arr2` to live on independently. This example ends up making two copies, one for each initialization.

The term **copy semantics** refers to the rules that determine how copies of objects are made. When we say a type supports copy semantics, we mean that objects of that type are copyable, because the rules for making such copies have been defined. When we say copy semantics are being invoked, that means we've done something that will make a copy of an object.

For class types, copy semantics are typically implemented via the copy constructor (and copy assignment operator), which defines how objects of that type are copied. Typically this results in each data member of the class type being copied. In the prior example, the statement `std::vector arr2 { arr1 };` invokes copy semantics, resulting in a call to the copy constructor of `std::vector`, which then makes a copy of each data member of `arr1` into `arr2`. The end result is that `arr1` is equivalent to (but independent from) `arr2`.

When copy semantics is not optimal

Now consider this related example:

```
#include <iostream>
#include <vector>

std::vector<int> generate() // return by value
{
    // We're intentionally using a named object here so mandatory copy elision
doesn't apply
    std::vector arr1 { 1, 2, 3, 4, 5 }; // copies { 1, 2, 3, 4, 5 } into arr1
    return arr1;
}

int main()
{
    std::vector arr2 { generate() }; // the return value of generate() dies at the
end of the expression

    // There is no way to use the return value of generate() here
    arr2[0] = 7; // we only have access to arr2

    std::cout << arr2[0] << '\n';

    return 0;
}
```

When `arr2` is initialized this time, it is being initialized using a temporary object returned from function `generate()`. Unlike the prior case, where the initializer was an lvalue that could be used in future statements, in this case, the temporary object is an rvalue will be destroyed at the end of the initialization expression. The temporary object can't be used beyond that point. Because the temporary (and its data) will be destroyed at the end of the expression, we need some way to get the data out of the temporary and into `arr2`.

The usual thing to do here is the same as in the previous example: use copy semantics and make a potentially expensive copy. That way `arr2` gets its own copy of the data that can be used even after the temporary (and its data) is destroyed.

However, what makes this case different than the previous example is that the temporary is going to be destroyed anyway. After initialization is complete, the temporary doesn't need its data any more (which is why we can destroy it). We don't need two sets of data to exist simultaneously. In such cases, making a potentially expensive copy and then destroying the original data is suboptimal.

Introduction to move semantics

Instead, what if there was a way for `arr2` to "steal" the temporary's data instead of copying it? `arr2` would then be the new owner of the data, and no copy of the data would need to be made. When ownership of data is transferred from one object to another, we say that data has been **moved**. The cost of such a move is typically trivial (usually just two or three pointer assignments, which is way faster than copying an array of data!).

As an added benefit, when the temporary was then destroyed at the end of the expression, it would no longer have any data to destroy, so we wouldn't have to pay that cost either.

This is the essence of **move semantics**, which refers to the rules that determine how the data from one object is moved to another object. When move semantics is invoked, any data member that can be moved is moved, and any data member that can't be moved is copied. The ability to move data instead of copying it can make move semantics more efficient than copy semantics, especially when we can replace an expensive copy with an inexpensive move.

Key insight

Move semantics is an optimization that allows us, under certain circumstances, to inexpensively transfer ownership of some data members from one object to another object (rather than making a more expensive copy).

Data members that can't be moved are copied instead.

How move semantics is invoked

Normally, when an object is being initialized with or assigned an object of the same type, copy semantics will be used (assuming the copy isn't elided).

However, when all of the following are true, move semantics will be invoked instead:

- The type of the object supports move semantics.
- The initializer or object being assigned from is an rvalue (temporary) object.
- The move isn't elided.

Here's the sad news: not that many types support move semantics. However, `std::vector` and `std::string` both do!

We'll dig into how move semantics works in more detail in chapter 22. For now, it's enough to know what move semantics is, and which types are move-capable.

We can return move-capable types like `std::vector` by value

Because return by value returns an rvalue, if the returned type supports move semantics, then the returned value can be moved instead of copied into the destination object. This makes return by value extremely inexpensive for these types!

Key insight

We can return move-capable types (like `std::vector` and `std::string`) by value. Such types will inexpensively move their values instead of making an expensive copy.

Such types should still be passed by const reference.

Wait, wait, wait. Expensive-to-copy types shouldn't be passed by value, but if they are move-capable they can be returned by value?

Correct.

This following discussion is optional, but may help you understand why this is the case.

One of the most common things we do in C++ is pass a value to some function, and get a different value back. When the passed values are class types, that process involves 4 steps:

1. Construct the value to be passed.
2. Actually pass the value to the function.
3. Construct the value to be returned.
4. Actually pass the return value back to the caller.

Here's an example of the above process using `std::vector`:

```
#include <iostream>
#include <vector>

std::vector<int> doSomething(std::vector<int> v2)
{
    std::vector v3 { v2[0] + v2[0] }; // 3 -- construct value to be returned to
caller
    return v3; // 4 -- actually return value
}

int main()
{
    std::vector v1 { 5 }; // 1 -- construct value to be passed to function
    std::cout << doSomething(v1)[0] << '\n'; // 2 -- actually pass value

    std::cout << v1[0] << '\n';

    return 0;
}
```

First, let's assume `std::vector` is not move-capable. In that case, the above program makes 4 copies:

1. Constructing the value to be passed copies the initializer list into `v1`
2. Actually passing the value to the function copies argument `v1` into function parameter `v2`.
3. Constructing the value to be returned copies the initializer into `v3`
4. Actually returning the value to the caller copies `v3` back to the caller.

Now let's talk about how we might optimize the above. We have many tools are our disposal here: pass by reference or address, elision, move semantics, and out parameters.

We can't optimize copies 1 and 3 at all. We need a `std::vector` to pass to the function, and we need a `std::vector` to return -- these objects have to be constructed. `std::vector` is an owner of its data, so it necessarily makes a copy of its initializer.

What we can affect is copies 2 and 4.

Copy 2 is made because we're passing by value from the caller to the called function. What other options do we have?

- Can we pass by reference or address? Yes. We are guaranteed that the argument will exist for the entire function call -- the caller does not have to worry about the passed object unexpectedly going out of scope.
- Can this copy be elided? No. Elision only works when we're making a redundant copy or move. There's no redundant copy or move here.

- Can we use an out parameter here? No. We're passing a value to the function, not getting a value back.
- Can we use move semantics here? No. The argument is an lvalue. If we moved data from `v1` to `v2`, `v1` would become an empty vector, and subsequently printing `v1[0]` would lead to undefined behavior.

Clearly pass by const reference is our best option here, as it avoids the copy, avoids null pointer issues, and works with both lvalue and rvalue arguments.

Copy 4 is made because we're passing by value from the called function back to the caller. What other options do we have here?

- Can we return by reference or address? No. The object we're returning is created as a local variable inside the function, and will be destroyed when the function returns. Returning a reference or pointer will result in the caller receiving a dangling pointer or reference.
- Can this copy be elided? Yes, possibly. If the compiler is smart, it will realize that we're constructing an object in the scope of the called function and returning it. By rewriting the code (under the as-if rule) so that `v3` is constructed in the scope of the caller instead, we can avoid the copy that would otherwise be made when returning. However, we are reliant upon the compiler realizing it can do this, so it is not guaranteed.
- Can we use an out parameter here? Yes. Instead of constructing `v3` as a local variable, we can construct an empty `std::vector` object in the scope of the caller, and pass it to the function by non-const reference. The function can then fill this parameter with data. When the function returns, this object will still exist. This avoids the copy, but also has some significant downsides and constraints: ugly calling semantics, doesn't work with objects that don't support assignment, and it is challenging to write such functions that can work with both lvalue and rvalue arguments.
- Can we use move semantics here? Yes. `v3` is going to be destroyed when the function returns, so instead of copying `v3` back to the caller, we can use move semantics to move its data to the caller, avoiding the copy.

Elision is the best option here, but whether it happens is out of our control. The next best option for move-capable types is move semantics, which can be used in cases where the compiler doesn't elide the copy. And for move-capable types, move semantics is invoked automatically when returning by value.

To summarize, for move-capable types, we prefer to pass by const reference, and return by value.