C++ - Understand copy elision

Apr 5, 2020 • Razakhel

In C++, there is a subtle mechanism which can avoid copies (and even moves), namely the "copy elision". As this expression suggests, this means that a copy can be avoided in certain cases.

[Named] Return Value Optimization

Given the following code:

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

int i = foo();
```

The (simplified & naive) stack view would technically look like this without copy elision:

int i;	**int foo() { ... }**	**{ return 42; }**	**i = foo();**
...	...	...	...
	foo() (int) (?)	foo() (int) (42)	
i (int) (?)	i (int) (?)	i (int) (?)	i (int) (42)

The returned value is first initialized, then copied into the variable we need.

With copy elision, the variable we need is *directly filled* by the call:

int i ...	**... = foo();**	**int foo() { return 42; }**
...	...	...
i (int) (?)	i/foo() (int) (?)	i/foo() (int) (42)

Both memory locations, at the call site and inside the function, are linked, so that filling inside the function the value to be returned actually operates on the one we return to.

With an int, that doesn't make much of a difference. However, keep in mind that **this works for any type**:

```
class Foo {
public:
    Foo() { std::cout << "Default constructor" << std::endl; }

    Foo(const Foo&) { std::cout << "Copy constructor" << std::endl; }
    Foo(Foo&&) { std::cout << "Move constructor" << std::endl; }

    Foo& operator=(const Foo&) {
        std::cout << "Copy assignment operator" << std::endl;
        return *this;
    }
    Foo& operator=(Foo&&) {
        std::cout << "Move assignment operator" << std::endl;
        return *this;
    }

    ~Foo() { std::cout << "Destructor" << std::endl; }
};

Foo getFooRVO() {
    return Foo();
}

Foo getFooNRVO() {
    Foo foo;
    return foo;
}

int main() {
    {
        std::cout << "--- RVO" << std::endl;
        Foo foo = getFooRVO();
    }

    {
        std::cout << "\n--- NRVO" << std::endl;
        Foo foo = getFooNRVO();
    }
}
```

```
--- RVO
Default constructor
Destructor

--- NRVO
Default constructor
Destructor
```

(Live example)

Note that I've used two functions for this: `fooRVO()` and `fooNRVO()`. Those are two cases of copy elision, and mean respectively "Return Value Optimization" and "*Named* Return Value Optimization".

The difference here is that `getFooRVO()` returns a value directly instantiated, and `getFooNRVO()` returns a variable (hence the "named" specification).

RVO actually isn't considered as copy elision anymore, since it is *guaranteed* in C++17[1] (and was commonly applied before anyway). NRVO isn't, since it can't be used in some particular cases, but is applied whenever possible.

Why not make use of move semantics?

As stated in the previous section and as its name implies, copy elision allows removing copy operations... but not just that. If you read carefully the previous example, you can see that *no move has been made*. Copy elision allows to bypass both copy *and* move operations.

To explain what introducing move semantics would do, let's take a quick look at C++ value categories.

Value categories

There are two main categories in C++, which you've probably seen earlier: lvalues and rvalues. The following explanation, although not perfectly accurate, hopefully will help to make the distinction clear:

- lvalues are variables which are not xvalues (see below). They were given their name from the fact that they were technically values on the left side of the equal sign on an assignation (*left* values).
- rvalues, contrary to lvalues, have their name originating from the fact that they were on the right side of the equal sign on an assignation (*right* values). rvalues are separated in two sub-categories:
  - prvalues (*pure* rvalues) are values which either are a literal (like `42` or `3.523`), an object construction, or a returned value from a function call.

- xvalues (*expiring* values) are lvalues that have been "transformed" to rvalues, by the application of an std::move() for example.

```
int i = 42;
// 42 is a prvalue, i is an lvalue

std::string str = "string";
// "string" is an lvalue (special case of string literals), str is an lvalue

std::string movedStr = std::move(str);
// str is changed into an rvalue (by becoming an xvalue), movedStr is an lvalue

std::string assignatedStr = std::string("str");
// std::string("str") is a prvalue, assignatedStr is an lvalue

std::string getString() { return "str"; }
std::string returnedStr = getString();
// The returned value from getString() is a prvalue, returnedStr is an lvalue
```

In the last two examples, copy elisions are performed. Those operations each result in a single construction of the string; neither move nor copy are performed.

I've stated earlier that RVO is guaranteed in C++17 and has been available prior to that. This is *an advantage gained by prvalues*: the compiler can effectively make use of the fact that prvalues are actually available in-place, and so is able to remove any move or copy operation. prvalues are not temporary values, they are literally *values themselves*, and as such can be propagated easily anywhere.

To get a more detailed and thorough explanation on value categories, see the dedicated page on cppreference.

Move semantics *can* hurt performance

Ok, that title may sound a little overdramatic. But it says "can", not "does", so everything's not lost!

We've said earlier that applying an std::move() creates an xvalue. Let's recall its definition: "xvalues are **lvalues** that have been transformed to rvalues".

What this means is that a value *must exist at some point as an lvalue* (or in other words, "must have a memory address") so that an xvalue can be created from it.

Taking the same example as above, let's add another function which moves out its returned value, and 3 usage examples:

```
// ...

Foo getFooMove() {
    return std::move(Foo());
}

int main() {
    {
        std::cout << "--- Move return" << std::endl;
        Foo foo = getFooMove();
    }

    {
        std::cout << "\n--- Move assignation" << std::endl;
        Foo foo = std::move(Foo());
    }

    {
        std::cout << "\n--- Move return & assignation" << std::endl;
        Foo foo = std::move(getFooMove());
    }
}
```

```
--- Move return
Default constructor
Move constructor
Destructor
Destructor

--- Move assignation
Default constructor
Move constructor
Destructor
Destructor

--- Move return & assignation
Default constructor
Move constructor
Destructor
Move constructor
Destructor
Destructor
```

(Live example)

The output has changed: *a move construction has been performed*. This is because the returned value from `getFooRVO()` has been transformed to an xvalue, thus losing its possible optimization as a prvalue. Likewise, one more destruction has occurred, since a temporary object has been created. The result is the same whether the value is moved on the return clause or at the call site.

In the third output, when std::move() is applied both when returning the value *and* when retrieving it, without surprise, twice the operations are applied.

As these results demonstrate, moving a returning value inside or outside a function can prevent the compiler from applying valuable optimizations[2].

Compiler to the rescue

We've said earlier that NRVO is not guaranteed to be applied. One notable example is when the returned value depends on a condition:

```
Foo getFooConditional(bool test) {
    if (test) {
        Foo foo;
        return foo;
    } else {
        return Foo();
    }
}

int main() {
    {
        std::cout << "--- NRVO failed" << std::endl;
        Foo foo = getFooConditional(true);
    }
}
```

```
--- NRVO failed
Default constructor
Move constructor
Destructor
Destructor
```

Notice that even in this case, where NRVO is not applicable, *a move has still been performed*[3]. It doesn't imply a copy if the type is allowed to be moved. The compiler will always try to move the value by itself, or then, if impossible, copy it.

What do you think would happen if we were to take the other branch? `return Foo();` is clearly the same as the RVO example, should it behave the same way?

```
int main() {
    {
        std::cout << "--- RVO success" << std::endl;
        Foo foo = getFooConditional(false);
    }
}
```

```
--- RVO success
Default constructor
Destructor
```

As a matter of fact, it does. Even within conditional branches, RVO is always mandatory.

(Live example)

Leave the compiler alone!

Even if a move may be better than a copy, it still isn't free; copy elision is. There is no point trying to explicitly move values while they actually do even better than that on their own.

There *are* cases where moving a value in a return clause is legitimate (namely, when the value comes from outside the function and is returned back). However, by systematically moving values coming from the inside of a function or when getting its result, you're actually forcing the compiler to do things that will perform worse than what it would have done on its own.

Compilers do a **much** better job than humans in figuring out what to do with your code, let them do what they know best. Don't bother trying to outperform them, while you can actually undermine them by doing so.

---

1. Being guaranteed means that it is safe, in a function, to return an instance of a class which has its copy & move constructors deleted. ↩
2. Note that both GCC & Clang have a warning dedicated to detect this kind of issues, `-Wpessimizing-move` (included in `-Wall`), which in this case is triggered by Clang on every attempt to move values. ↩
3. The application of the NRVO is highly dependent on the compiler. On the same example, Clang actually avoids the move operation even within the branch. However, instantiating the returned variable out of the condition forces Clang to move it when returning. ↩

Razakhel's blog

Romain Milbert
romain.milbert@gmail.com

Razakhel
romainmilbert

Programming blog, revolving around C++ and RaZ, my own game engine.