On harmful overuse of std::move

devblogs.microsoft.com/oldnewthing/20231124-00

November 24, 2023

Raymond Chen

The C++ `std::move` function casts its parameter to an rvalue reference, which enables its contents to be consumed by another operation. But in your excitement about this new expressive capability, take care not to overuse it.

```
std::string get_name(int id)
{
    std::string name = std::to_string(id);
    /* assume other calculations happen here */
    return std::move(name);
}
```

You think you are giving the compiler some help by saying "Hey, like, I'm not using my local variable `name` after this point, so you can just move the string into the return value."

Unfortunately, your help is actually hurting. Adding a `std::move` causes the `return` statement to fail to satisfy the conditions for copy elision (commonly known as Named Return Value Optimization, or NVRO): The thing being returned must be the name of a local variable with the same type as the function return value.

The added `std::move` prevents NVRO, and the return value is move-constructed from the `name` variable.

```
std::string get_name(int id)
{
    std::string name = std::to_string(id);
    /* assume other calculations happen here */
    return name;
}
```

This time, we return `name` directly, and the compiler can now elide the copy and put the `name` variable directly in the return value slot with no copy. (Compilers are permitted but not required to perform this optimization, but in practice, all compilers will do it if all code paths return the same local variable.)

The other half of the overzealous `std::move` is on the receiving end.

```
extern void report_name(std::string name);

void sample1()
{
    std::string name = std::move(get_name());
}

void sample2()
{
    report_name(std::move(get_name()));
}
```

In these two sample functions, we take the return value from `get_name` and explicitly `std::move` it into a new local variable or into a function parameter. This is another case of trying to be helpful and ending up hurting.

Constructing a value (either a local variable or a function parameter) from a matching value of the same type will be elided: The matching value is stored directly into the local variable or parameter without a copy. But adding a `std::move` prevents this optimization from occurring, and the value will instead be move-constructed.

```
extern void report_name(std::string name);

void sample1()
{
    std::string name = get_name();
}

void sample2()
{
    report_name(get_name());
}
```

What's particularly exciting is when you combine both mistakes. In that case, you took what would have been a sequence that had no copy or move operations at all and converted it into a sequence that creates two extra temporaries, two extra move operations, and two extra destructions.

```
#include <memory>

struct S
{
    S();
    S(S const&);
    S(S &&);
    ~S();
};

extern void consume(S s);

// Bad version
S __declspec(noinline) f1()
{
    S s;
    return std::move(s);
}

void g1()
{
    consume(std::move(f1()));
}
```

Here’s the compiler output for msvc:

```
; on entry, rcx says where to put the return value
f1:
    mov     qword ptr [rsp+8], rcx
    push    rbx
    sub     rsp, 48
    mov     rbx, rcx

    ; construct local variable s on stack
    lea     rcx, qword ptr [rsp+64]
    call    S::S()

    ; copy local variable to return value
    lea     rdx, qword ptr [rsp+64]
    mov     rcx, rbx
    call    S::S(S &&)

    ; destruct the local variable s
    lea     rcx, qword ptr [rsp+64]
    call    S::~S()

    ; return the result
    mov     rax, rbx
    add     rsp, 48
    pop     rbx
    ret

g1:
    sub     rsp, 40

    ; call f1 and store into temporary variable
    lea     rcx, qword ptr [rsp+56]
    call    f1()

    ; copy temporary to outbound parameter
    mov     rdx, rax
    lea     rcx, qword ptr [rsp+48]
    call    S::S(S &&)

    ; call consume with the outbound parameter
    mov     rcx, rax
    call    consume(S)

    ; clean up the temporary
    lea     rcx, qword ptr [rsp+56]
    call    S::~S()

    ; return
    add     rsp, 40
    ret
```

Notice that calling `g1` resulted in the creation of a total of two extra copies of `S`, one in `f1` and another to hold the return value of `f1`.

By comparison, if we use copy elision:

```
// Good version
S __declspec(noinline) f2()
{
    S s;
    return s;
}

void g2()
{
    consume(f2());
}
```

then the msvc code generation is

```
; on entry, rcx says where to put the return value
f2:
    push    rbx
    sub     rsp, 48
    mov     rbx, rcx

    ; construct directly into return value (still in rcx)
    call    S::S()

    ; and return it
    mov     rax, rbx
    add     rsp, 48
    pop     rbx
    ret

g2:
    sub     rsp, 40

    ; put return value of f1 directly into outbound parameter
    lea     rcx, qword ptr [rsp+48]
    call    f2()

    ; call consume with the outbound parameter
    mov     rcx, eax
    call    consume(S)

    ; return
    add     rsp, 40
    ret
```

You get similar results with gcc, clang, and icc icx.

In gcc, clang, and icx, you can enable the `pessimizing-move` warning to tell you when you make these mistakes.