# Last use optimization

Document: P2666R0

Date: 2022-10-10

Project: Programming language C++

Audience: EWG(I)

Reply-to: Bengt Gustafsson, bengt.gustafsson@beamways.com



## Introduction

In previous C++ standard versions more and more optimizations of return values have been added, with acronyms like RVO and NRVO. These optimizations reduce the number of copy constructions and move constructions occurring in return statements by not performing copy and move operations exactly as it seems from the source code.

This proposal generalizes these rules to any code when the compiler can prove that a value used as argument of a function call is not going to be used after the function returns regardless of how control flows. This includes the cases where you today correctly wrote `std::forward` and `std::move` but also the places *where you forgot* to do so. In summary the values considered are:

- Local by-value variables
- By-value parameters
- Rvalue reference parameters
- Local references referring to local variables which contain last-used values.
- Dereferenced Local pointers which can be proven to point at local variables which contain last-used values.
- Non-static data members inside rvalue qualified functions.

Last use of the value stored in one of these ways refers to not only to locations where the compiler can deduce that the variable is not at all used later, but also that the *next use* is definitely an assignment of a new value. This covers the important case of modifying a value like in `name = toUpper(name);` where the old contents of name is allowed to be moved into toUpper as the next use of name is it being assigned the result of toUpper.

In addition this proposal includes rules that mandate this optimization in simple cases, to allow programmers to omit move/forward calls in those cases.

# Motivation and scope

Adding `std::forward` and `std::move` calls in all places where it is correct can get tedious and obscures the intent of the code. It is also very easy to forget to add these function calls, and in some cases when control flow is complicated it can be hard to manually determine if a variable has possible uses afterwards or not, opening up for bugs related to over-use of forward and move. Furthermore there is a teaching problem in understanding when to use move and when to use forward.

We have not done a quantitative analysis of how many percent of moves and forwards are missed in general code, but we suspect that this is a fairly high ratio, especially in functions whose main purpose is not to just forward their parameters but where a last use of a variable or parameter is as an argument to a function which has lvalue and rvalue overloads. Here is an example of a function where the string `row` would cause an unnecessary allocation in push_back.

```
void add_member(const std::string& first, const std::string& last)
{
    auto row = std::format("First name: {}, Last name: {}", first, last);
    members.push_back(row);     // Missed move opportunity here!
}
```

In addition this optimization will be possible in cases where the same variable is used more than once in the same full expression, where forwarding can't portably be used today due to unknown order of evaluation. For instance:

```
template<typename T1, typename T2> auto concat(T1&& lhs, T2&& rhs) {
    return std::forward<T1>(lhs) + std::forward<T2>(rhs);
}

std::string twice(std::string word)
{
    return concat(word, word);
}
```

In this case the programmer can't reliably move any of the uses of `word` as it is unknown which one will be evaluated first. A compiler will know its own order of evaluation of parameter expressions and can perform a move in the correct position to preserve the semantics.

The rules apply to the implicit object parameter of member functions: If the last use of a parameter is to call a member function with an rvalue overload, it will be called.

```
struct A {
    void f() const  & { std::cout << "lvalue f"; }
    void f() const && { std::cout << "rvalue f"; }
};

void g(A&& a) {
    a.f();  // LUO applies here
}
```

# Impact on the standard

This proposal affects the standard by allowing demoting copy to move in more places. The wording involved should match the corresponding wording for return value optimizations, but involves a writing like "when the compiler can determine that the variable being moved is never used again, or when its next use is being assigned a new cvalue".

One complicating factor is that the compiler needs more logic to detect suitable sites for LUO which is less obvious than for RVOs where it is in return expressions only. Further discussion below.

## Side effect difference risks

The main backwards compatibility risk is that the program behaviour can change if user defined move and copy constructors have (different) side effects. This is exactly the same concerns that one could raise against RVO and yet RVO in its different varieties were introduced in the language.

Please note that although RVO is mainly about not copying/moving at all it is also allowed to demote a copy to a move in some situations since C++11, and the number of situations when this is allowed is increased in C++23.

We think that with RVO as precedent these risks are possible to accept. In real life the most common case will probably be that the move constructor is buggy but by chance it was never used before this optimization was introduced. This is the same situation as for RVO and in particular the increased number of situations being introduced in C++23.

Here are a couple of examples where the behaviour would change:

```
class A {
    A() : m(42);
    A(const A& src) : m(src.m) { std::cout << "copy" << std::endl; }
    A(A&& src) { std::cout << "buggy move" << std::endl; }

    HeavyType m;
}:

void f(A&&) { std::cout << " rvalue f" << std::endl; }
void f(const A&) { std::cout << " lvalue f" << std::endl; }

A g(A a, A b, A&& c, A&& d) {
    A e = a;   // Before/after LUO:     "copy"/"buggy move"
    f(b);      // Before/after LUO:     "lvalue f"/"rvalue f"
    f(c);      // Before/after LUO:     "lvalue f"/"rvalue f"
```

```
    return d;  // Before/after P1825R0: "copy"/buggy move"
}
```

Here the optimization allows the construction of e to call the move constructor and the calls of f to call the rvalue overload. In C++20, due to the lack of explicit `std::move` calls, e is copy-constructed from a and the `const A&` overload of f is called.

The optimization triggers the latent bug that the member m is not handled by the move constructor, which is an example of the risks with this proposal.

However, these types of risks were considered when P1825 was accepted for C++20, as exemplified by the `return d` which after C++20 started forgetting to set m member of the returned object.

## Dual use risk

Going back to the concat example we notice another kind of risk.

```
std::string twice(std::string word)
{
    return concat(word, word);
}
```

Here the previously explained rule would treat the last evaluated word instance as a rvalue. This works for concatenations of strings thanks to the careful design of the different overloads. But maybe the implementers of `concat` didn't think about the prospect of being called with two references to the same string. In C++20 this works *thanks to* the undefined evaluation order, which disallows programmers from moving any of the uses of word.

Now the claim is that *thanks to* the compiler knowing the evaluation order it can treat whichever parameter that gets evaluated last as an rvalue as it is the last use. This changes which overload of `concat` that gets called and may break code that previously worked. To clarify, here is a concat for strings that does not work:

```
MyString concat(const MyString& lhs, const MyString& rhs);  // #1
MyString concat(const MyString& lhs, MyString&& rhs);       // #2
MyString concat(MyString&& lhs, const MyString& rhs);       // #3

MyString w1 = "w1";
auto ww = concat(w1, w1);              // #4
auto ww1 = concat(w1, std::move(w1));  // #5

MyString w2 = "w2";
auto ww2 = concat(std::move(w2), w2);  // #6
```

In C++20 #4 is guaranteed to call the first concat overload which works. No careful programmer would dare write #5 or #6 given the unspecified evaluation order so #2 and #3 go untested for the same parameter case. With this new optimization overload #2 or #3 will get called, depending on the parameter evaluation order chosen by the compiler and if they fail to handle that both parameters are the same the code siltently breaks.

One way of avoiding this risk is to only allow converting the last use of a variable in an expression where it is used more than once if there is only one by reference parameter referring to the variable. Thus in this case, as there are two by reference parameters no optimization can be done.

## Mandatory LUO rule

To allow programmers to wittingly omit `std::forward` and `std::move` in their code and rely on the compiler to handle it for them there must be a minimum requirement on compilers to detect the last use of a variable in simple cases that portable code can rely on. An appropriate level of mandatory analysis must be standardized, with the trade off being between the ease of implementation in compilers and the code complexity where programmers can trust the compiler to find the last uses. However, pushing this too far may be counter-productive as programmers would get a hard time figuring out if they have to write a `std::forward` or `std::move` themselves and may put too high hopes on the compiler doing it for them.

We recommend a mandatory LUO rule where the compiler must find all last uses outside of loops, when the function does not contain goto statements. This means that not only the lexically last use must be found but also uses which are in side by side if-else constructs or switch cases as long as there is no surrounding loop which would allow control flow to reach the same site again. Furthermore last uses of non-static data members are found using the same rules for rvalue-qualified functions. The mandatory rule does not include any indirect use of a variable via a references or dereferenced pointers, or cases when the next use is unambiguously an assignment to the variable.

This example reuses the A class and f functions from the previous example.

```
void g()
{
    A a, b, c, d, e;
    f(a);

    if (condition...) {
        f(c);
        f(b);
    }
    else
        f(b);

    while (true) {
        A n;
        f(d);

        if (phase_of_moon() == New) {
            f(e);
            return;
        }
        f(n);
        break;
    }
}
```

Here the calls with a, b, c and n would all be mandatory LUO while the use of d and e is not mandatory. A smart compiler would notice that d and e are also last uses, and that in the case of e this would be the case even if the last break was removed. The reason that n is mandatory although it is inside a loop is that it is both declared and last used inside the loop, i.e. a new instance of n is created in each loop turn.

If this level of control flow analysis is considered too complex to require from all compliant compilers a lower level would be to only mandate LUO outside of all control structures in the scope of the variable being used. This still finds most uses and should be nearly trivial to implement. In the example this would mean that only a and n would invoke mandatory LUO.

### ABI impact

We don't think that this proposal has any impact on ABI as it only changes which function is called, not how the selected function is called. The only exception would be a pre-compiled library with a header file that claims that a copy constructor is present, but where there is no implementation available.

### Compile time impact

As this feature potentially applies to all call sites it is important to analyse what impact introducing this feature may have on compile times. This could affect on the decision regarding the mandatory LUO rule selected. While modern compilers do advanced data flow analysis this is done mainly in release builds and may be done at a stage after overload resolution. There are levels that are definitely lightweight such as only considering the lexical position in the scope of declaration when determining which use is last (excluding cases where the last use is in any form of loop).

### Interaction with relocation

This proposal interfers constructively with relocation proposals in that the possible sites for relocations are the move sites. If the compiler can find more potential move sites it can also demote these further, to relocations, if allowed. This requires further exploration, for instance a next assignment after a move turned relocation would have to be changed from assignment to placement new.

## Technical specification

Before any wording attempt we need direction for the mandatory LUO rule. For the other parts we think that text relating to RVO can be reused to some extent, complemented by text relating to the compiler detecting that a use is a last use of a variable and implicitly converting the variable from lvalue to rvalue before overload resolution of the function call takes place.

## Future directions

### Increasing the envelope of mandatory demotion

As for RVO it is possible to introduce this feature piece-meal with less and less restrictions, increasing the complexity of code where programmers can rely on the compiler doing the forwarding for them. This increase of scope can occur along two axes:

- Increasing the complexity of code where the compiler is oblighed to find move opportunities.

- Increasing the number of constructs which are eligible, such as local references, dereferenced pointers and variables that are reassigned.

## Eliding moves

One future direction that has not yet been fully understood is the possibilities for eliding moves altogether. If and when this is possible for function calling depends to a large extent on the ABI calling conventions.

We think that the move eliding possibilities are unrelated to whether the ABI is caller-destroy or callee-destroy. In the latter case the compiler would however have to keep track of the fact that the local variable pointed to has already been destroyed when the called function returns. This would disrupt the reverse order of destruction.

We therefore think that eliding moves should be a separate proposal, where the possible impact of the out of order destruction is thoroughly analysed.

# Acknowledgements

Thanks to my employer ContextVision AB for supporting the author attending standardization meetings.

Thanks to Björn Andersson and David Friberg of the Swedish national body for valuable feedback and new and improved examples.