Analysis of interaction between relocation, assignment, and swap

P3278R0

Author: Nina Ranns
Corentin Jabot
Date: 21/05/2024
Audience: LEWG, EWG
Project: ISO/IEC 14882 Programming Languages — C++, ISO/IEC JTC1/SC22/WG21

Abstract

In this paper, we analyze the relationship between relocation, assignment, and swap. Additionally, we address the points raised in P3236.

Introduction

P2786 and P1144 both attempt to address relocation and trivial relocation for the purposes of optimization. Both papers also define the meaning of relocation in pretty much identical ways, although P2786 goes further in addressing the lifetime aspects of relocation:

From P1144
"We define a new verb, "relocate," equivalent to a move and a destroy. For many C++ types, this is implementable "trivially," as a single memcpy."

From P2786
"— To relocate a type from memory address `src` to memory address `dst` means to perform an operation or series of operations such that an object equivalent (often identical) to that which existed at address `src` exists at address `dst`, that the lifetime of the object at address `dst` has begun, and that the lifetime of the object at address `src` has ended."

Many operations within the standard library are morally relocations, such as vector::erase and vector::insert, yet they are often implemented in terms of assignment. However, we argue assignment and relocation are two different notions. We do want to enable optimisations of such operations when we know that the trivial relocation will do "the right thing".

Swap exchanges values of two existing objects, but can for a certain subset of types be implemented as a series of operations like memmove. Because trivial relocation is also implemented as a series of bit blasting operations, it is tempting to reuse relocation as one way of optimizing swap. However, it is important to note that swap isn't a relocation and that additional considerations need to be made when optimizing swap in terms of bit blasting operations.

Relocation vs Assignment

It is reasonable to expect that all operations that morally perform relocation should be able to benefit from trivial relocation optimisations, according to some definition of trivially relocatable types. It is quite common to view operations like `vector::erase` and `vector::insert` as performing relocations. However, the standard specification is opaque about whether these operations are relocating whole objects or just managing values.

In practice, most standard library implementations use assignment when moving values to locations where there are currently objects. In some cases, this is even implied with both requirements for assignability and statements like this one :

"Complexity: The number of calls to the destructor of T is the same as the number of elements erased, but the number of calls to the assignment operator of T is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements"

In general, we have not so far distinguished between construction and assignment. Both were used to move values around. With the introduction of relocation, we get another aspect of operation that we have not considered so far - what happens with the old object. Relocation is inherently a construction of a new object and destruction of the old object. In assignment, there is no expectation on what happens to the old object.

This distinction on what happens to the old object when moving values matters in certain cases
We can imagine an object that holds some trivially relocatable ID and a value. On assignment, the value is exchanged without affecting the ID. However, relocation should result in an object with the same ID. A type holding such an ID will need a user defined assignment operator in order to only assign the parts of the object that make up the value.
On construction of such an object, if we can guarantee that the old object will be destroyed, we can reuse the ID. If no such guarantee exists, the ID can not be reused.
Because ID and all parts of this type are relocatable, it makes sense for the type to be relocatable too. After all, destruction of the old object **is** a part of relocation. An author of such type may want to mark their type as relocatable.

This brings us back to operations which are “morally” relocations. Any new implementations of such operations can use trivial relocation for types that are deemed trivially relocatable, and should be allowed to do so for types for which relocation can be trivially, including those types for which assignment yields a different result to relocation. P3236 recognises the problem of already existing implementations that use assignment as a part of the implementation of a relocation, and the desire to keep the current semantics. We can provide the ability to do so without unnecessarily pessimizing all relocating operations. The missing bit is the ability to distinguish between what happens on assignment vs what happens on relocation.
As it turns out, the same is needed in order to optimize swap.

Relocation vs swap

When talking about swap, the standard talks about swapping *values*. For certain subset of types, a value is also the object representation. If value and object representation are the same, a swap can also be implemented as a set of bit blasting operations. This discussion only resolves the question on which types can have optimized swap. The second part of optimizing swap is addressing lifetime considerations.

Ignoring lifetime considerations, two things need to hold for swap to be able to be reduced to bit blasting operations
1) the type needs to be trivially relocatable
2) the semantics should not change if we use bit blasting operations instead of assignment

It’s already been observed that the latter is the same as what is needed in order to optimize current relocating operations which use assignment in the implementation. However, the current approaches conflate the notion of implicit trivial relocatability with the notion of “trivial relocation can be used for assignment” and consequently “trivial relocation can be used for swap”. With P1144’s definition of implicit trivial relocation, one can guarantee that both #1 and #2 hold for implicitly trivially relocatable types, but the same is not true for the types which are explicitly marked as trivial relocatable.

We need to recognise one additional key difference between a relocation operation (i.e. vector::insert) and swap:

- In a relocation operation, the source object is destroyed after relocation
- In swap, both objects are alive after the swap.

The consequence of this is that, regardless of how we define what an implicitly trivially relocatable type is, we still need to have a way of knowing whether an explicitly marked trivially relocatable type has semantics that allow us to implement swap in terms of trivial relocation.

Consider the type with a trivially relocatable ID from earlier on. We want objects of such type to be relocated within vector::erase and vector::insert so we mark them as trivially relocatable.

When an object of such type is relocated, the ID persists because we know that the source object will be destroyed and we can't end up in a situation where two objects hold the same ID. Now consider a swap of two objects of such type. The type is designed in such a way that assignment only exchanges the values, but does not affect the ID of the object. A swap is expected to exchange the values, but not modify the ID of the object. Despite the fact the object is marked as trivially relocatable, we can't optimize swap to use relocation because the guarantee that relocation provides (i.e. the source object is destroyed after relocation) no longer holds. This problem can not be solved by simply reflection on trivial relocatability of a type, even if we make user provided assignment disable implicit trivial relocation because the user is allowed to mark their type explicitly trivially relocatable, and this type is arguably able to be relocated when relocations guarantees hold.

Having established that swapping the value representation of types is a distinct operation from relocation, and given that std::swap has additional considerations that must be taken into account, we need a separate facility to enable the optimization of swap. That optimization is important but should not be conflated with relocation, even if both build on the notion of trivially relocatable types.

We suggest that the way forward is to carve out types for which construction+destruction is equivalent to assignment, but we do not offer any concrete solutions. For the lack of better naming, we refer to these types as replaceable types (P3239 - calls this swap_uses_value_representations types). Implementing optimised swap would then be a case of writing a specialization for types which are trivially relocatable and replaceable. Note that the notion of replacable could also be used in other situations where one needs to understand whether assignment has different semantics to copy construction, like deciding when to optimize relocation operations that are currently implemented using assignment. This would also solve the problems related to `tuple<int&>` raised by P3236 as one could make tuple<int&> trivially relocatable, but not replaceable..
We refer the reader to P3239 , which aside from proposing a way to optimize swap, also offers a way of specifying which subset of trivially relocatable types can have optimized swap, namely by the mean of a type trait that indicates whether swapping the value representation of a type is isomorphic to a call to std::swap. We do not offer an opinion as to whether the specifics of the design proposed by P3239 are the right approach, we are simply emphasizing the need for separation of the notion of relocation from the notion of swap.

Lifetime considerations

P3239 recognises additional problems with swap that need to be resolved regardless of how we determine the subset of types which can have an optimized swap. The issues stem from the fact

that bit blasting operations like memmove affect object lifetimes. Trivial relocation by definition affects lifetimes of source and destination. However, swap itself is not meant to affect the lifetimes of the objects being swapped. To implement swap in terms of either trivial relocation or, more specifically, in terms of a series of bit blasting operations, we need to create a way of doing so that does not end the lifetimes of the objects being swapped. P3239 suggests a magic function that swaps the objects using relocation, but does not end the lifetime of either of the objects.

Virality of trivial relocatability

When allowing users to explicitly mark their type as relocatable, there are several options

1) check whether all the subobjects of such type are relocatable, and prevent marking the top level type as relocatable if some parts are not relocatable
2) allow the author to mark the top level as relocatable only if all of the subobjects are not explicitly marked as not relocatable
3) allow the author to mark the top level as relocatable in all cases

Options 2 and 3 allow for an easier transition. However, option 1 is the safest option. We can consider relaxing the rules at a later point to allow for more cases when a type can be marked as relocatable, but we can not make the rules more strict.

We should note that any design along the lines of option 2 and 3 are not maintainable over time or even libraries because the properties of the type of the subobjects might change inadvertently. Maybe there is a desire for an easy way to succinctly express trivial relocatability conditionally based on the properties of subobjects, but we feel that this is a rare situation (most types are either trivially relocatable implicitly or are not relocatable at all), and these rare uses cases do not warrant weakening the ability of the compiler to check the soundness of trivial_relocatable specifiers.

Note that the question of when to allow a type to be marked as trivially relocatable based on the trivial relocatibility of its subobjects is a separate question to other ones posed in this paper and should be considered separately.

Attribute vs new syntax

P3236 argues that an attribute would allow enabling the feature in earlier language modes.

However, EWG reaffirmed many times that attributes should be ignorable and as such not have semantic impact.
In general, trivial relocation is both a contractual property of a type, and something that can be observed through traits. For example, trivial relocatable types can be moved across address spaces in the context of GPU/heterogenous programming (which doesn't imply their subobject are safe to use), which is a property the compiler must preserve and enforce.

`no_unique_address` and the subsequent decision by some implementations not to adopt it caused significant pains in the ecosystem, and we should try not to repeat the same mistakes. Some implementers (notably Clang's) expressed their opposition to an attribute syntax for trivial relocation. Following both EWG's and Implementers guidance, P2786R5 chose not to bring up the question of attribute ignorability again, preferring introducing a new contextual keyword.

We should note however that implementers can (and likely will) backport this feature in older language modes (in part so that standard libraries can support relocation unconditionally).
All implementations warn on unknown attributes such that using an attribute would not permit using trivial relocation with older toolchains without the use of a macro.

Note that the question of whether we should have an attribute or a core language feature to mark a type as relocatable is a separate question to all the other questions posed in this paper and should be considered separately.

Conclusion

Recognising the difference between relocation and assignment is imperative in solving the problem of introducing relocation into the standard. Having a way of reflecting on whether we can replace assignment with relocation, in addition of being able to reflect whether a type is trivially relocatable, offers the most benefit:

- new relocation operations are not limited by the assignment semantics of the type
- library implementors have a choice on choosing which way to optimise for already existing relocation operations, allowing them not to pessimize based on assignment operator
- swap can be correctly implemented
- we have a mental model that does not conflate the notion of relocation and assignment

We should also come to an explicit agreement on all the questions raised in this paper, regardless of which relocation paper we go for. While some of these points were discussed in EWG session in Tokyo 2024, no explicit polls were taken. The questions to explicitly agree on are (at least):

- Should we use a core feature or an attribute to mark a type as trivially relocatable?
- Should implicit trivial relocatibility be affected by the assignment operator ?

- What should be the restrictions on types which are explicitly marked as trivially relocatable?

Additionally, we need to address the problem of lifetimes when using relocation in swap. The only paper we have that currently addresses this problem is P3236.

Acknowledgments

A big thank you to Alisdair Meredith, Joshua Berne, Mungo Gill, Pablo Halpern, and Ville Voutilainen who volunteered their time and energy to discuss the topic of relocation at length.