Relocator: Efficiently moving objects

by denis bider, 2016-04-08 *Current email always at: www.denisbider.com*

Problem

Current C++ implements move construction, but without providing the promise that the object being moved from will be destroyed. This prevents efficient movement of objects in a number of scenarios, primarily impacting containers, which would prefer to perform a minimal cost relocation.

As currently practiced by a number of libraries[1], relocation involves undefined behavior where an object's bytes are copied to a new memory location, and the object's type contract is resumed at the new location, without a destructor or constructor being called. This has problems in that the object transfer is implied rather than explicit, which is a headache for static analysis; does not accommodate objects that require fix-ups; and relies on error-prone user declarations.

Non-nullable types

Prime examples of objects which cannot be efficiently content-moved are objects that maintain non-null invariants. For example, a move constructor for a non-null pointer type[2]; or an `std::list` implementation that maintains a non-null invariant when empty[3]; must allocate a new instance of data to be left behind in the object that's being moved from. This means:

- The type is not `nothrow_move_constructible`. A `vector` wanting to provide strong exception safety cannot use the object's move constructor during resize, insert, or erase, but must perform a potentially expensive deep copy.
- Even if the `vector` forgoes exception safety, and uses the move constructor, the move constructor performs work that the destructor immediately undoes. Temporary objects are created and then immediately destroyed.
- When a non-nullable type is stored in a `variant`, the variant cannot provide exception safety for assignment and emplace operations, and a "valueless" variant state is introduced.

General container use

Even objects that have a non-throwing move constructor, and a non-throwing destructor, cannot be moved as efficiently as relocation would permit. For a majority of types, a relocation would be equivalent to `memcpy`. For a minority of types, it would be equivalent to `memcpy` with an additional step to fix-up self-references. For example, fix-ups are needed by `std::string` in libstdc++, which uses self-references for short string optimization; yet they are not needed by `std::unique_ptr`, or `std::string` in Visual Studio.

For that majority of object types where a relocation would be equivalent to `memcpy`, a formally correct object array move that calls the move constructor, followed by the destructor, can be up to 5 – 10 times slower than `memcpy`. The slowdown is due to the overhead of calling a destructor unnecessarily, as well as branching in the destructor which serves to determine that there are no resources to free.

[1] **Qt** (Q_MOVABLE_TYPE), **EASTL** (has_trivial_relocate), **BSL** (IsBitwiseMovable), **Folly** (IsRelocatable)
[2] Which e.g. DropBox "uses pervasively" (not associated with this author): https://github.com/dropbox/nn
[3] Pablo Halpern – Destructive Move, N4034: http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4034.pdf

Proposal

Relocator

We define a special member function, a *relocator*, with the following example syntax:

```
class A : B {
  >>A(A& x) : >>B(x)     // Defer to base relocator
            , >>c(x.c)   // Defer to member relocator
            , >>i(x.i)   // Defer to trivial member relocator
    { a = this; }        // Fix up a self-reference. It is only this
                         //   self-reference that requires an explicit relocator
  C c;
  int i;
  A* a;
};
```

This is a cross between a move constructor and a destructor that performs both tasks, and thus has full access to the object in the old location. The relocator needs a non-const reference in case it wishes to call a subobject destructor (see *Fallback to different types of construction*), or to zero-out storage.

For most types, a relocator will be trivial, and can be defaulted:

```
class A : B {
  A();
  A(A const& x);        // Blocks implicit declaration of relocator
  >>A(A&) = default;    // Defers to >>B(B&), >>C(C&), >>int(int&)
  C c;
  int i;
};
```

Noexcept guarantee

All relocators are implicitly `noexcept`. Declaring a potentially throwing relocator is illegal.

Fallback to different types of construction

This proposal permits uses such as the following – a variation of the above example:

```
>>A(A& x) : B(std::move(x)), >>c(x.c), >>i(x.i) { x.B::~B(); a = this; }
```

In place of relocation, the user may choose to initialize a base or a member using move construction from the original subobject; or by calling a different, unrelated constructor on the subobject being initialized.

In this case, since the subobject is not being relocated, the user is responsible for invoking the destructor on the original subobject. The syntax shown – `x.B::~B()` – requires no new support from the language, and limits destruction only to B and its bases and members, even if the `~B` destructor is virtual.

A relocator relying on such methods must still meet the `noexcept` guarantee.

Direct invocation

For a single object, the user invokes relocation using syntax similar to placement new:

```
// void* d – destination (uninitialized), T* s – source (initialized)

T* dt = new (d) >>T(*s);

// d now points to initialized memory; s to uninitialized
```

For object arrays, the user can invoke relocation using a standard `std::relocate`, behaving as follows:

```
// Multiple objects, trivially relocatable

template <class T, enable_if_t<is_trivially_relocatable_v<T>, int> = 0>
T* relocate(void* d, T* s, size_t n) noexcept {
  memmove(d, s, n*sizeof(T));
  return (T*) d;
}

// Multiple objects, not trivially relocatable

template <class T,
   enable_if_t<is_relocatable_v<T> && !is_trivially_relocatable_v<T>, int> = 0>
T* relocate(void* d, T* s, size_t n) noexcept {
  T* dt = (T*) d;
  if (dt < s) {      // Support overlapping memory for container insert/erase
    for (size_t i=0; i!=n; ++i)
      new (dt+i) >>T(s[i]);
  }
  else if (dt > s) {
    while (n--)
      new (dt+n) >>T(s[n]);
  }                  // Else, dt == s; do nothing
  return dt;
}
```

The proposed `std::relocate` does not bundle move construction in order to avoid penalizing users who do not wish such functionality bundled. Instead, this proposal suggests additional functions:

```
template <class T> T* relocate_or_move(void* d, T* s);
template <class T> T* relocate_or_move(void* d, T* s, size_t n);
```

These functions choose relocation or move, as available. Suggested definitions are provided in *Annex 1*.

Indirect invocation via copy/move optimization

When a type is relocatable; then in situations where an implementation would otherwise copy/move an object value to an object of same type by invoking a copy/move constructor, followed by destruction of the moved-from object; and where no further use of the moved-from object occurs before destruction; the implementation **should** instead invoke the type's relocator, eliding the copy/move and destruction.

This optimization **must** occur when copy/move followed by destruction would otherwise be used when returning a relocatable type by value from a function, into a newly constructed object of same type.

Annex 2 proposes a syntax allowing a user to ensure that when move construction or relocation is intended, it is in fact used. When an implementation cannot conclusively determine whether an object may still be accessed, this syntax would allow the user to provide a hint and a diagnostic requirement via `std::must_relocate`. If the user requests relocation in this indirect manner, but the language implementation conclusively determines that this is unsafe, the implementation must generate an error.

Object lifetime and invocation limits

To the extent a relocator does not explicitly destroy subobjects, lifetime of the complete original object continues until the outermost relocator completes, at which point it is considered to have ceased.

Invoking a subobject relocator outside of complete object relocation results in undefined behavior.

Implicit relocator

Declared as defaulted

If the definition of a class X does not explicitly declare a relocator, a non-explicit one is implicitly declared as defaulted, if and only if class X satisfies the following criterion for each other special member:

- X does not have a user-declared *(special member)*, or the user-declared *(special member)* is defaulted at first declaration.

This criterion must be satisfied for the following special members: copy constructor; move constructor; copy assignment operator; move assignment operator; and destructor.

This is relaxed from existing rules for implicit declaration of a move constructor, which are stricter in that a move constructor will not be implicitly declared if any special member is user-declared, *even* if all such special members are defaulted at first declaration.

Rules for implicit declaration of copy/move constructors are further amended so that if a class has a user-declared relocator that is not defaulted at first declaration, no copy/move constructors can be implicitly declared as defaulted.

Defined as deleted

A defaulted relocator for a class X is defined as deleted if X has any potentially constructed subobject with a relocator that is deleted, not declared, or inaccessible from the defaulted relocator.

Implicitly defined

A relocator that is defaulted and not defined as deleted is implicitly defined if it is odr-used or when it is explicitly defaulted after its first declaration.

Before the defaulted relocator for a class is implicitly defined, all non-user-provided relocators for potentially constructed subobjects of its type shall have been implicitly defined.

An implicitly defined relocator defers to relocators of its direct bases and members.

Type traits

In the header `type_traits`, new properties are defined as follows:

```
template <class T> struct is_relocatable;
template <class T> struct is_trivially_relocatable;
```

The value of `is_relocatable<T>::value` is true if T has either a user-defined relocator, or a defaulted relocator that is not defined as deleted.

The value of `is_trivially_relocatable<T>::value` is true if T has a trivial relocator. A trivial relocator is one that is defaulted, not deleted, and calls only other trivial relocators. It is equivalent to `memcpy`.

Impact on existing code

Under the proposed rules, plain-old-data types that:

- declare no non-defaulted special members (copy/move constructor/assignment; destructor);
- contain no subobjects that do;

receive an implicit relocator, declared as defaulted and trivially implementing a memberwise copy. This is safe, because objects of such types are trivial, and can already be copied and moved via `memcpy`.

For classes that declare one or more non-defaulted special members: no implicit relocator is declared, and there is no change of behavior.

For classes that:

- declare no non-defaulted special members;
- but contain subobjects that do;

there is a change in behavior if, and only if, all such subobjects provide accessible relocators. In this case, an implicit relocator is declared as defaulted, and if odr-used, is implicitly defined. This is expected to be innocuous, since an object defined this way is already delegating copy and move concerns to its subobjects. A slightly stricter version of this rule has worked for implicitly defaulted move constructors.

Discussion

Move constructor/destructor vs. constructing destructor vs. destructing constructor

I settled on the name “relocation” – as opposed to “destructive move”, “destructing construction”, or “constructing destruction” – because the latter versions intuitively suggest an entangled operation that incurs the costs of both construction and destruction; whereas, this is a lightweight operation that simply relocates objects in memory, allowing for minor patch-ups.

The name should reflect that the operation is neutral in terms of resource freeing and acquisition. In this sense, “relocation” seems less prone to misleading implications.

No zombies

In this proposal, relocators are invoked through a type-clean interface that converts destination storage from uninitialized to initialized; and the source storage vice versa. There are no zombie objects in undefined states left behind – beyond uninitialized storage, as it is already left behind by destructors.

Why not destructor-like syntax?

Original drafts of this proposal suggested destructor-like syntax, as follows:

```
~A(A& x) : ~B(x), ~c(x.c), ~i(x.i) { x.a = &x; }
```

This would operate very similarly to the current proposal, but with the possibly unintuitive difference that the new object is being passed as the parameter, so that e.g. `~c(x.c)` initializes `x.c` with the value of `c`. Furthermore, it would require re-inventing syntax to allow for move construction of subobjects.

Why not a const reference?

If the relocator accepted a const reference, this would suggest that the relocator cannot change original memory, or that such changes may lead to undefined behavior. This would impede usage such as:

- a relocator that zeroes out data stored by the object in its previous location;
- a relocator that move constructs a subobject, and therefore must call the destructor of the original copy of the subobject. For example:

```
>>A(A& x) : B(std::move(x)), >>c(x.c) { x.B::~B(); }
```

Why not an rvalue reference?

If the relocator accepted an rvalue reference, this might convey more obviously that the relocator will destroy the object in its previous location. However, it would require syntax such as the following:

```
>>A(A&& x) : >>B(std::move(x)), >>c(std::move(x.c)) {}   // Not being suggested
```

An lvalue is more convenient, and there is no overloading conflict to solve with an rvalue reference.

Prior work

This proposal can be seen as building on prior ideas, some of which I was aware of at the time of writing; and some of which I was not, but they reached me through other people regardless.

In May 2014, Pablo Halpern submitted N4034 – Destructive Move, proposing type traits to be explicitly declared by class implementers, and used by a standard function to provide a destructive move:

http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4034.pdf

Also in May 2014, Sean Parent published a Non Proposal for Destructive Move, contemplating a move destructor using rvalue syntax, focusing on use cases outside of containers:

https://github.com/sean-parent/sean-parent.github.io/wiki/Non-Proposal-for-Destructive-Move

Summary of opportunity

Relocation improves the performance of container insert, erase, and resize operations for all objects.

Maintaining a non-null invariant can be a useful way to design an object, such as an `std::list`, that avoids explicit checks and branching each time an object is accessed. Without relocation, the desire for efficient use with containers, based on existing content move semantics, drives us to design objects with null resting states, even when we would not otherwise choose this. This leads us to another instance of what has been called the Billion-Dollar Mistake[4] – except this time, with object state instead of pointers.

This proposal would allow non-nullable types to be moved efficiently, with strong exception safety. It would allow `std::variant` to be used with non-nullable types without introducing a valueless state.

Annex 3 discusses the problem of the valueless `std::variant`, and how relocation would avoid it.

In the absence of a `realloc_in_place`, containers cannot safely take advantage of `realloc` in C++ as-is. `realloc_in_place` would be preferable for type-safe relocation of both trivially and non-trivially relocatable types. However, in the absence of a `realloc_in_place`, libraries that wish to offend authors of static analysis tools could safely use `realloc` for types with trivial relocators.

Annex 4 includes a more in-depth discussion of opportunities in the area of in-place reallocation.

Thanks to:

- **David Rodríguez Ibeas**, for alerting me to the idea of a special destructive-move member function as a move destructor, in an exchange in the ISO C++ std-proposals forum, August 5, 2015;
- **Matthew Woehlke**, for valuable feedback to the "Move destructor" draft, as well as later drafts;
- Reddit users in http://www.reddit.com/r/cpp :-)
- **Vitali Lovich**, for further feedback, and the suggestion to explore constructor-like syntax;
- **Edward Catmur**, for suggestions about interaction with move construction;
- **Thiago Macieira**, for the suggestion to make the relocation wrapper a standard library function.
- **Sean Middleditch**, **Barry Revzin**, and others, for pointing out the need to formalize indirect use.

[4] Tony Hoare – "Null References: The Billion Dollar Mistake": http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Annex 1: std::relocate_or_move

In addition to `std::relocate`, which this proposal suggests to perform relocation only, several commenters have expressed a desire for a standard facility that bundles relocation and move+destroy, and uses the best option available. A suggested implementation follows:

```
template <class T, enable_if_t<is_relocatable_v<T>, int> = 0>
T* relocate_or_move(void* d, T* s) noexcept {
  return new (d) >>T(*s);
}

template <class T,
  enable_if_t<!is_relocatable_v<T> &&
              is_nothrow_move_constructible_v<T> &&
              is_nothrow_destructible_v<T>, int> = 0>
T* relocate_or_move(void* d, T* s) noexcept {
  T* dt = new (d) T(move(*s));
  s->~T();
  return dt;
}

template <class T, enable_if_t<is_relocatable_v<T>, int> = 0>
T* relocate_or_move(void* d, T* s, size_t n) noexcept {
  return relocate(d, s, n);
}

template <class T,
  enable_if_t<!is_relocatable_v<T> &&
              is_nothrow_move_constructible_v<T> &&
              is_nothrow_destructible_v<T>, int> = 0>
T* relocate_or_move(void* d, T* s, size_t n) noexcept {
  T* dt = (T*) d;
  if (dt < s) {
    for (size_t i=0; i!=n; ++i) {
      new (dt+i) T(move(s[i]));
      s[i].~T();
    }
  }
  else if (dt > s) {
    while (n--) {
      new (dt+n) T(move(s[n]));
      s[n].~T();
    }
  }
  return dt;
}
```

Versions supporting copy construction; or potentially throwing move construction or destruction; are not suggested, because they cannot provide identical guarantees and behavior. For example, such versions cannot provide a strong exception safety guarantee if destination and source memory overlap, as is the case in container erase or insertion.

Annex 2: Ensuring use of intended initialization types

Current C++ syntax makes it difficult for a user to ensure that, when move construction is intended, it is in fact being used. As code changes, a move constructor may become silently unavailable. For example:

```
BigPart MakeBigPart() {
  BigPart part;
  ...
  return part;     // may elide, move, or copy
}

std::unique_ptr<BigThing> MakeBigThing() {
  auto part = MakeBigPart();
  auto thng = std::make_unique<BigThing>(std::move(part));   // may move or copy
  return thng;
}
```

If `BigPart` changes, the above may begin to invoke unintended deep copies. Relocation adds new failure modes of noexcept relocation being substituted with a deep copy, or with throwing move construction.

Instead of the user having to rely solely on their reasoning; with no diagnostic when the user is incorrect; a user may want to instruct the language implementation to use a particular initialization type. One way to do so might be with a family of special standard library functions:

```
template <class T> T&  must_relocate     (T&); // elide or relocate
template <class T> T&& must_noexcept_move(T&); // elide/relocate or noexcept move
template <class T> T&& must_move         (T&); // elide/relocate or move
template <class T> T&& must_noexcept_init(T&); // elide or any noexcept copy/move
```

Each of these would require the language implementation to flag the returned reference internally, and verify at compile-time that it either ends up passed to a matching initializer, or that instantiation is elided.

This would allow the above example to be changed as follows:

```
BigPart MakeBigPart() {
  BigPart part;
  ...
  return std::must_move(part);     // may elide, relocate, or move - but not copy
}

std::unique_ptr<BigThing> MakeBigThing() {
  auto part = MakeBigPart();
  auto thng = std::make_unique<BigThing>(std::must_move(part));  // relocate/move
  return thng;                     // do not care if copied
}
```

This would guarantee that copy construction of `BigPart` is not being triggered directly; although copy construction could still be invoked as part of move construction or relocation of a subobject.

Annex 3: Relocation as a solution for the valueless variant problem

The valueless variant problem has been recognized in other proposals, including P0308[5], which proposes "pilfering" as a possible solution. Relocation is an alternate solution which does not require pilfering.

The valueless variant problem arises if a move constructor throws during assignment or emplacement, as in the following example:

```
std::variant<X, Y> a = X();
std::variant<X, Y> b = Y();
try { b = std::move(a); }    // bad_alloc here
catch (std::bad_alloc) {}
```

In this case, `variant`'s assignment operator has to do something like this:

```
b.y.~Y();
new (&b.x) X(std::move(a.x));
```

If X's move constructor throws, `b` ends up with no value. Using a temporary does not avoid this:

```
X temp { std::move(a.x) };
b.y.~Y();
new (&b.x) X(std::move(temp));
```

The problem now arises if the second move constructor throws. However, with relocation, it *does* work:

```
X temp { std::move(a.x) };        // We use syntax in Annex 2 to relocate from an
b.y.~Y();                         // automatic object. Alternative: placement new
new (&b.x) X(std::must_relocate(temp));  // into std::aligned_storage, then >>X()
```

This now has strong exception safety, without introducing a valueless `variant` state. The same solution works for `emplace`: the object can be constructed in a temporary, and then noexcept-moved or relocated.

This does require `X` to implement a relocator if its move constructor throws. However, requiring a relocator is not like requiring a move constructor. A relocator for most any type can be implemented trivially, without changes to design. Most user types which do not get an implicit relocator because they declare a special member function may obtain one by declaring `>>Type(Type&) = default`.

Given the ease of implementing a relocator for most any type that can be moved at all, it seems plausible that the combination of a type with a throwing move constructor, which also lacks a relocator, may not need to be supported by new library types which do not have pre-existing compatibility requirements.

[5] Peter Dimov – Valueless Variants Considered Harmful, P0308R0: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0308r0.html

Annex 4: The need for in-place reallocation

This proposal stands on its own. However, it could provide further benefit if the C++ standard library adds allocator functionality equivalent to `realloc_in_place`. This is currently not available in the standard, but container implementers may already be able to take advantage of it using platform-specific functions.

The following is available on Windows. I expect similar functionality to be available on other platforms:

```
template <class T>
bool ReAllocInPlace (T* p, size_t n) noexcept
   { return HeapReAlloc(processHeap, HEAP_REALLOC_IN_PLACE_ONLY,
                        p, n*sizeof(T)) != nullptr; }
```

1. Under the assumptions that:

- `realloc_in_place`, or a platform-specific equivalent, is available; and
- `realloc` is a strict superset of `realloc_in_place`, with pseudo-code as follows:

```
if (!realloc_in_place) { malloc; memcpy; free; }   // realloc
```

Then the following combination, implied in this proposal:

- copying relocator with signature: `>>T(T&)`
- relocation strategy: `if (!realloc_in_place) { malloc; relocate; free; }`

... is strictly at least as efficient as; or more efficient than; the following alternate strategy:

- patching relocator with signature: `>>T(ptrdiff_t)`
- relocation strategy: `if (realloc(p) != p) fixup;`

This is because the former, with the copying relocator, boils down to:

```
if (!realloc_in_place) { malloc; relocate; free; }
```

Whereas the latter, with a patching relocator, boils down to:

```
if (!realloc_in_place) { malloc; memcpy; free; fixup; }
```

For types that require fix-ups – such as any type that contains a libstdc++ string – the patching strategy involves making two passes. Where memory access dominates, this may double the cost of relocation.

2. Additionally, when inserting objects into a container, `realloc_in_place` allows this:

```
if (!realloc_in_place) { malloc; relocate(part1); relocate(part2); free; }
insert;   // insert into space between part1 & part2 already created in relocation
```

Whereas when only `realloc` is available, it boils down to:

```
if (!realloc_in_place) { malloc; memcpy(part1+part2); free; fixup(part1+part2); }
memmove(part2); fixup(part2);   // additional steps to create space for insertion
insert;
```

It would be nice if containers could make use of these efficiencies using standard memory allocation.