Destructive Move | N4034

Pablo Halpern phalpern@halpernwightsoftware.com

2014-05-27

1 Abstract

This paper proposes a function template for performing *destructive move* operations – a type of move construction where the moved-from object, instead of being left in a "valid, but unspecified" state, is left in a destructed state. I will show that this operation can be made non-throwing in a wider range of situations than a normal move constructor, and can be used to optimize crucial operations, such as reallocations within vectors. An array version of the destructive move template is proposed specifically for moving multiple objects efficiently and with the strong exception guarantee.

The facilities described in this paper are targeted for a future library Technical Specification.

2 Motivation

2.1 Background

The main reason that rvalue references and move operations were introduced into the standard was to improve performance by reducing expensive copy operations. The `noexcept` keyword was added in order to support a number of important use cases where move operations could not otherwise be used. Because move constructors modify the moved-from object, an operation that moves multiple elements, e.g., in a container, could result in both containers being in a half-moved state if one of the move constructors throws an exception. It is not possible to reliably reverse this half-moved situation without risking another exception being thrown.

Using `noexcept`, an implementation can detect whether it is possible that a move constructor might throw and, if so, can choose copy construction, instead. In fact, the standard provides the function template `move_if_noexcept` specifically for this purpose. The following implementation of `push_back` for a simplified vector uses this idiom to preserve the *strong exception guarantee*, whereby the moved-from vector remains unchanged if an exception is thrown:

```
template <class T, class A = std::allocator<T>>
class simple_vec
{
    A           m_alloc;        // allocator to obtain space for elements
    T*          m_data;         // address of allocated storage (or null)
    std::size_t m_capacity;     // size (in elements) of allocate storage
    std::size_t m_length;       // number of elements in container

  public:
    ...
    void push_back(const T& v);
```

```
};

template <class T, class A>
void simple_vec<T, A>::push_back(const T& v)
{
    typedef std::allocator_traits<A> alloc_traits;

    if (m_length == m_capacity) {
        // Grow the vector by creating a new one and swapping.
        simple_vec temp(m_alloc);
        temp.m_capacity = (m_capacity ? 2 * m_capacity : 1);
        temp.m_data = alloc_traits::allocate(m_alloc, temp.m_capacity);

        T *from = m_data, *to = temp.m_data;
        for (temp.m_length = 0; temp.m_length < m_length; ++temp.m_length)
            alloc_traits::construct(m_alloc, to++,
                                    std::move_if_noexcept(*from++));
        temp.swap(*this);
        // Destructor for 'temp' destroys moved-from elements.
    }

    alloc_traits::construct(m_alloc, &m_data[m_length], v);
    ++m_length;
}
```

2.2 Lost opportunities

Unfortunately, the requirement that move constructors leave the moved-from object in a valid (though unspecified) state results in several important situations where a move constructor cannot be decorated with `noexcept`. For example, some implementations of `list`, including at least one commercial implementation, use a heap-allocated sentinel node in order to preserve the stability of the `end()` iterator when using `swap` and `splice`. A moved-from `list` implemented this way must have a sentinel node in order to avoid an "emptier than empty" violation of its class invariants. The default constructor and move constructor for such a list might look like the following (`m_begin` and `m_end` are member variables pointing to the first and past-the-end nodes in the list. `Node` is class representing a single list node.):

```
// default constructor for a simple list type (no allocator support)
template <class T>
simple_list<T>::simple_list()
{
    m_begin = m_end = new Node(nullptr, nullptr);  // might throw
}

// move constructor for a simple list type (no allocator support)
template <class T>
simple_list<T>::simple_list(simple_list&& other)
{
    simple_list temp;  // Default constructor might throw.
    temp.swap(*this);  // 'swap' never throws.
}
```

Since the sentinel node requires a memory allocation, which might throw, neither the default constructor nor the move constructor can be decorated with `noexcept`. Such a type cannot benefit from the `move_if_noexcept` optimization – it would need to be copied every time.

As you can see from the `push_back` code for `simple_vec` in the previous section, after all of the elements have been moved from one place to the other, the moved-from elements are destroyed. It is not necessary in this and many similar situations to leave the moved-from object in a valid state – it would be sufficient to end its lifetime as part of the move (i.e., as if its destructor had been called).

Why is this important? Many, if not most, classes that cannot offer a nothrow move constructor *can* offer a nothrow destructive move operation – a move combined with a destroy. In the case of our `simple_list`, above, the destructive move operation would simply move the `m_begin` and `m_end` pointers from the list being moved from to the list being moved to. Any attempt to use (or destroy) the moved-from object would be undefined behavior. Thus, a destructive move operation could expand the set of cases that could benefit from `move_if_noexcept`-like optimizations.

Another benefit of destructive move is that it is often more efficient to perform a destructive move operation than a non-destructive move construction. In the case of a string, for example, a destructive move would simply copy pointers. Since pointers are trivially copyable, the entire move operation becomes a trivial copy that can be implemented as a `memcpy`. This optimization is magnified when operating on arrays of strings: the entire array can be destructively moved with a single `memcpy`. This optimization was implemented at Bloomberg before move constructors were even invented and has yielded significant performance gains. It turns out that a large number of classes, like `string`, can be destructively moved using byte copies. Such classes model a concept I call *trivially destructive-movable*.

Note that trivially destructive-movable does not require or imply trivially copyable; unlike a copy, after the destructive move is complete, the moved-from object must not be accessed, since any pointer members would point to memory shared with the moved-to object.

3 Proposal summary

This proposal comprises two new function templates and two new traits. The first function template is called `destructive_move` and looks like this:

```
template <class T>
  void destructive_move(T* to, T* from) noexcept( /* see below */ );
```

The preconditions are that `from` points to a valid object and `to` points to raw memory. The postconditions are that `to` points to a valid object and `from` points to raw memory. The default implementation is simply:

```
::new(to) T(std::move(*from));
to->~T();
```

This default, however, can be overridden in two ways:

1. If the trait `is_trivially_destructive_movable<T>` is true, then the destructive move is implemented using byte copies. This trait is always true for types that are trivially movable, but can also be overridden for other types by class authors.
2. If `destructive_move` is overloaded for a specific type, then ADL will resolve to the overloaded version. Thus, a class author can implement an efficient destructive move (with its own `noexcept` clause) for his/her new type.

The `noexcept` specification for this function template is computed to be `true` for as many types as possible. Only if a type has a throwing move constructor and does not override the default for destructive move is the `noexcept` specification false. The `is_nothrow_destructive_movable<T>` trait is defined to be true if `destructive_move<T>` has a true `noexcept` specification.

Note, however, that destructive move cannot be used with an idiom like `move_if_noexcept`. The `move_if_noexcept` function guarantees that its argument remains valid whether an exception is thrown or not. The `destructive_move` function, in contrast, invalidates its second argument on success and leaves it unchanged if an exception is thrown. A hypothetical `destructive_move_if_noexcept` function could result in a third possible end state: both `to` and `from` could be valid if the object `is_nothrow_destructive_movable<T>` is false and the function were forced to make a copy. The usage idiom for such a function would be complex and non-intuitive. Instead, this paper proposes a function template `destructive_move_array`, which encapsulates the entire process of moving elements from one array to another and rolling back on failure. Using `destructive_move_array`, the implementation of `simple_vec::push_back` would look as follows:

```
template <class T, class A>
void simple_vec<T, A>::push_back(const T& v)
{
    typedef std::allocator_traits<A> alloc_traits;

    using std::experimental::destructive_move_array;

    if (m_length == m_capacity) {
        // Grow the vector by creating a new one and swapping
        simple_vec temp(m_alloc);
        temp.m_capacity = (m_capacity ? 2 * m_capacity : 1);
        temp.m_data = alloc_traits::allocate(m_alloc, temp.m_capacity);

        // Exception-safe move from this->m_data to temp.m_data
        destructive_move_array(temp.m_data, m_data, m_length);

        // All elements of 'temp' have been constructed and
        // all elements of '*this' have been destroyed.
        temp.m_length = m_length;
        m_length = 0;
        temp.swap(*this);
    }

    alloc_traits::construct(m_alloc, &m_data[m_length], v);
    ++m_length;
}
```

4 Formal wording for the TS

4.1 Header `<experimental/destructive_move>` synopsis

```
namespace std {
namespace experimental {
inline namespace fundamentals_vX {

  template <class T> struct is_trivially_destructive_movable;
  template <class T> struct is_nothrow_destructive_movable;

  template <class T>
    void destructive_move(T* to, T* from) noexcept( /* see below */ );

  template <class T>
```

```
    void destructive_move_array(T* to, T* from, size_t sz)
      noexcept(is_nothrow_destructive_movable<T>::value);

}
}
}
```

4.2 Type trait `is_trivially_destructive_movable`

```
namespace std {
namespace experimental {
inline namespace fundamentals_vX {

  template <class T>
  struct is_trivially_destructive_movable :
      integral_constant<bool, (is_trivially_move_constructible<T>::value &&
                               is_trivially_destructible<T>::value)>
  {
  };

}
}
}
```

A type `t` is *trivially destructive-movable* if, given two pointers to `T`, `p1` and `p2`, where `p1` points to an existing object that is not a base-class subobject and `p2` points to allocated storage of suitable size and alignment for an object of type `T`, copying the underlying bytes from `*p1` to `*p2` has the same user-visible effect as move-constructing `*p2` from `*p1` then destroying `*p1`. [*Note:* A type need not be trivially move-constructible nor trivially destructible in order to be trivially destructive-movable – *end note*]

The `is_trivially_destructive_movable` template shall be a UnaryTypeTrait with a base characteristic of `true_type` if it can be shown that `T` is *trivially destructive-movable*, otherwise `false_type`. [*Note:* False negatives are acceptable, but false positives would result in undefined behavior. – *end note*] A program may specialize `is_trivially_destructive_movable` for a user-defined class `T`. Such a specialization shall meet all of the requirements for this template.

4.3 Type trait `is_nothrow_destructive_movable`

```
namespace std {
namespace experimental {
inline namespace fundamentals_vX {

  template <class T> struct is_nothrow_destructive_movable;

}
}
}
```

The `is_nothrow_destructive_movable` template shall be a UnaryTypeTrait with a base characteristic of `true_type` if the expression `destructive_move<T>(p1, p2)` is known not to throw exceptions for valid arguments `p1` and `p2`.

4.4 Function template `destructive_move`

```
template <class T>
  void destructive_move(T* to, T* from) noexcept( /* see below */ );
```

Requires: If `is_trivially_destructive_movable<T>::value` is false, then `T` shall be MoveConstructible.

Preconditions: `to` shall be a pointer to allocated memory of suitable size and alignment for an object of type `T`; `from` shall be a pointer to an existing object that is not a base-class subobject.

Effects: If `is_trivially_destructive_movable<T>::value` is true, then equivalent to `memcpy(to, from, sizeof(T));` otherwise, equivalent to `::new(static_cast<void*>(to), move(*from)); from->~T();`. [*Note:* Overloads of this function for user-defined or library types may achieve the same postconditions by other means. – *end note*]

Throws: nothing unless the move constructor or destructor for `T` throws. The expression within the `noexcept` clause is equivalent to `is_trivially_destructive_movable<T>::value || (is_nothrow_move_constructible<T>::value && is_nothrow_destructible<T>::value)` Overloads of this function for specific types may have different exception specifications.

Postconditions: `*to` (after the call) is equivalent to (i.e., substitutable for) `*from` before the call except that it has a different address. The lifetime of `*from` is ended (but `from` still points to allocated storage). [*Note:* To avoid invoking the destructor on the destroyed object, `from` should not point to an object with automatic storage duration. – *end note*]

4.5 Function template `destructive_move_array`

```
template <class T>
  void destructive_move_array(T* to, T* from, size_t sz)
    noexcept(is_nothrow_destructive_movable<T>::value);
```

Requires: `T` shall be MoveConstructible. If `is_nothrow_destructive_movable<T>::value` is false, then `T` shall also be CopyConstructible and the destructor for `T` shall not throw for any element in `from`.

Preconditions: `to` shall be a pointer to allocated memory of suitable size and alignment for an array of `sz` elements of type `T`, `from` shall be a pointer to an existing array of `sz` elements of type `T`.

Effects: Constructs copies of the elements in `from` into the memory pointed to by `to` and destroys the elements in `from`. If `is_nothrow_destructive_movable<T>::value` is true, the copies are constructed as if by `destructive_move`, otherwise by copy construction.

Throws: Nothing unless the copy constructor for `T` throws. If an exception is thrown, the call shall have no effect.

5 Implementation Experience

Source code for the traits and function templates proposed in this paper, as well as an implementation of `simple_vec` and a test driver for the whole thing, is available at http://halpernwightsoftware.com/WG21/destructive_move.tgz. The code is free to use and distribute for both commercial and non-commercial purposes. Destructive move with specializations for trivially destructive-movable types has been in use in Bloomberg LP's BDE code base for well over 5 years and has resulted in significantly faster `vector` operations.

6 Future work

The `destructive_move` function template can be useful not only for non-overlapping operations such as vector reallocations, but also for overlapping array operations such as inserting and erasing elements. However, `destructive_move_array` is not suited to those overlapping moves. There is an opportunity to add one or two additional function templates for this purpose. Additionally, there may be use cases for a variant of `destructive_move_array` that would work on arbitrary iterator ranges rather than specifically on arrays.

7 Acknowledgments

Thanks to my former colleagues at Bloomberg for encouraging me to write this paper and reviewing early drafts.

8 References

BDE *BDE Library*, developed by Bloomberg LP. Source code available at https://github.com/bloomberg/bde

N3908 *Working Draft, Technical Specification on C++ Extensions for Library Fundamentals*, Jeffrey Yasskin, editor, 2014-03-02