DEEEER0 draft 3: SG14 `[[move_relocates]]`

Document #:	DEEEER0 draft 3
Date:	2018-04-16
Project:	Programming Language C++ Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

This proposes a new C++ attribute `[[move_relocates]]` which enables more aggressive optimisation of moves of those types than is possible at present.

A major motivation behind this proposal is to enable the standard lightweight throwable `error` object as proposed by [P0709] *Zero-overhead deterministic exceptions* to directly encapsulate a `std::exception_ptr`, using CPU registers alone for storage.

Changes since draft 2:

- Replaced the compiler deducements with a simple C++ attribute named `[[move_relocates]]` and reduced down the size and complexity of paper considerably to the bare essentials.

Changes since draft 1:

- Added requirement for default constructor, move constructor and destructor to be defined in-class.
- Added halting problem avoidance requirements.
- Refactored text to clearly indicate that trivial relocatability is just an optimisation of move.
- Mentioned STL allocators and effects thereupon.

Contents

1 Introduction **2**

1.1 Prior work in this area 3

2 Impact on the Standard **3**

3 Proposed Design **3**

4 Design decisions, guidelines and rationale **6**

5 Technical specifications **6**

6 Frequently asked questions **6**

7 Acknowledgements **6**

8 References 7

1 Introduction

The most aggressive optimisations which the C++ compiler can perform are to types which meet the `TriviallyCopyable` requirements:

- Every copy constructor is trivial or deleted.
- Every move constructor is trivial or deleted.
- Every copy assignment operator is trivial or deleted.
- Every move assignment operator is trivial or deleted.
- At least one copy constructor, move constructor, copy assignment operator, or move assignment operator is non-deleted.
- Trivial non-deleted destructor.

All the integral types meet `TriviallyCopyable`, as do C structures. The compiler is thus free to store such types in CPU registers, relocate them in memory as if by `memcpy`, and overwrite their storage as no destruction is needed. This greatly simplifies the job of the compiler optimiser, making for tighter codegen, faster compile times, and less stack usage, all highly desirable things.

There are quite a lot of types in the standard library and in user code which do not meet `TriviallyCopyable`, yet are completely safe to be stored in CPU registers and can be relocated arbitrarily in memory as if by `memcpy`. For example, `std::vector<T>` likely has a similar implementation to:

```
template<class T> class vector
{
  T *_begin{nullptr}, *_end{nullptr};
public:
  vector() = default;
  vector(vector &&o) : _begin(o._begin), _end(o._end) { o._begin = o._end = nullptr; }
  ~vector() { delete _begin; _begin = _end = nullptr; }
  ...
};
```

Such a vector implementation could be absolutely stored in CPU registers, and arbitrarily relocated in memory with no ill effect via the following as-if sequence:

```
vector<T> *dest, *src;

// Copy bytes of src to dest
memcpy(dest, src, sizeof(vector<T>));

// Copy bytes of default constructed instance to src
vector<T> default_constructed;
memcpy(src, &default_constructed, sizeof(vector<T>));
```

This paper proposes a new C++ attribute `[[move_relocates]]` which tells the compiler and library code when the move of a type with non-trivial move constructor and non-trivial destructor can be instead performed using two as-if `memcpy()`'s.

1.1 Prior work in this area

- [N4034] *Destructive Move*

  This proposal differs from destructive moves in the following ways:

  - It is considerably simpler than N4034, we do not propose any new kind of operation, nor new semantics. We merely propose an alternative way of doing moves, valid under limited circumstances.

- [P0023] *Relocator: Efficiently moving objects.*

  This proposal differs from relocators in the following ways:

  - It is considerably simpler than P0023, we do not propose any new kind of operation, nor new operators. We merely propose an alternative way of doing moves, valid under limited circumstances.

2 Impact on the Standard

Very limited. This is a limited optimisation of the implementation of move construction, and the STL move construct-destruct cycle only. We do not fiddle with allocators, the meaning nor semantics of moves, nor anything else.

All we propose is that where it is safe to do so, we replace the calling of the move constructor with `memcpy()` (which can be elided by the compiler if it has no visible side effects, same as with all `memcpy()`). That in turn enables temporary storage in CPU registers, if the compiler chooses to do so.

3 Proposed Design

1. That a new C++ attribute `[[move_relocates]]` become applicable to type definitions.
2. This attribute shall be silently ignored if the type does not have a public, non-deleted, constexpr, in-class defined default constructor and if there is not a public, non-deleted, non-virtual move constructor.
3. If a type `T` has attribute `[[move_relocates]]`, instead of calling the defined move constructor, the compiler will implement move construction as-if by `memcpy(dest, src, sizeof(T))`, followed by as-if `memcpy(src, &T{}, sizeof(T))`. Note that by 'as-if', we mean that the compiler can fully optimise the sequence, including the elision of calling the destructor if the

destructor would do nothing when supplied with a default constructed instance, which in turn would elide entirely the second memory copy.

4. It is considered good practice that the move constructor be defaulted with an explanatory comment mentioning the `[[move_relocates]]`, as the move constructor is never called on types with non ignored `[[move_relocates]]`. For backwards compatibility with older compilers, it is acceptable to implement the move constructor to cause the exact same effects as `[[move_relocates]]` i.e. copying the bits of source to destination followed by copying the bits of a default constructed instance to source.

5. If a type `T` has attribute `[[move_relocates]]`, the trait `std::is_relocatable<T>` shall be defaulted to true. Note that the programmer may specialise `std::is_relocatable<T>` to any value they like for some type `T`.

6. If STL containers see that for their type `T` that `std::is_relocatable<T>` is true, that `std::has_virtual_destructor<T>` is false, and if the Allocator they are configured with has a defaulted `construct()` and `destroy()`, they will relocate the storage for type `T` in the move construction + destruction cycle by a method equivalent to copying bits, but without calling the move constructor, nor the destructor on the moved-from storage.

   This introduces an important corner case: the programmer is free to write a non-virtual destructor for a `[[move_relocates]]` type which when called on a default constructed instance is non-trivial, however it will not be called when the type is used inside a STL container with the default Allocator.

7. `[[move_relocates]]` types with a non-virtual destructor are eligible for being automatically treated as having a trivial ABI[1] i.e. can be passed between function in CPU registers if the compiler can easily deduce[2] that the destructor, when called on a default constructed instance, would become trivial.

Let us take a worked example. Imagine the following partial implementation of `unique_ptr`:

```
template<class T>
class [[move_relocates]] unique_ptr
{
  T *_v{nullptr};
public:
  // Has public, non-deleted, constexpr default constructor
  unique_ptr() = default;

  constexpr explicit unique_ptr(T *v) : _v(v) {}

  unique_ptr(const unique_ptr &) = delete;
  unique_ptr &operator=(const unique_ptr &) = delete;

#if __cplusplus >= 202300
  // Has a public, non-deleted, move constructor
  unique_ptr(unique_ptr &&) = default;  // implemented via [[move_relocates]]
#else
```

[1] https://clang.llvm.org/docs/AttributeReference.html#trivial-abi-clang-trivial-abi

[2] 'Easily deduce' is actually quite hard to implement. A compiler might simply insist on there also being a `[[trivial_abi]]` attribute on the type, if it supported such an attribute.

```
  constexpr unique_ptr(unique_ptr &&o) noexcept : _v(o._v)
  {
    o._v = nullptr;
  }
#endif
  unique_ptr &operator=(unique_ptr &&o) noexcept
  {
    delete _v;
    _v = o._v;
    o._v = nullptr;
    return *this;
  }
  ~unique_ptr()
  {
    delete _v;
    _v = nullptr;
  }
};
```

The default constructor is not deleted, constexpr and public and it sets the single member data `_v` to `nullptr`. Additionally, the move constructor is not deleted, not virtual and public, so `[[move_relocates]]` is not ignored.

The destructor, when called on a default constructed instance, is trivial (`operator delete` does nothing when fed a null pointer, and setting a null pointer to a null pointer leaves the object with exactly the same memory representation as a default constructed instance).

We thus get, for this program:

```
unique_ptr<int> __attribute__((noinline)) foo()
{
  return unique_ptr<int>(new int);
}

int main()
{
  auto a = foo();
  return 0;
}
```

... that current C++ compilers will generate the following x64 assembler:

```
foo(): # @foo()
  push rbx
  mov rbx, rdi
  mov edi, 4
  call operator new(unsigned long)
  mov qword ptr [rbx], rax
  mov rax, rbx
  pop rbx
  ret
main: # @main
  push rax
  mov rdi, rsp
  call foo()
```

```
  mov rdi, qword ptr [rsp]
  test rdi, rdi
  je .LBB2_2
  call operator delete(void*)
.LBB2_2:
  xor eax, eax
  pop rcx
  ret
```

But under this proposal, the following x64 assembler would be generated instead:

```
foo(): # @foo()
  mov edi, 4
  jmp operator new(unsigned long) # TAILCALL
main: # @main
  push rax
  call foo()
  test rax, rax
  je .LBB2_2
  call operator delete(void*)
  xor eax, eax
.LBB2_2:
  pop rcx
  ret
```

Due to moves being known by the compiler to relocate as-if by `memcpy()`, the optimiser has spotted that `unique_ptr` can be elided entirely, and just the `new`ed pointer it contains is passed directly between functions, by CPU register.

4 Design decisions, guidelines and rationale

Previous work in this area has tended towards the complex. This proposal proposes the bare essentials for address relocatable types in the hope that the committee will be able to get this passed.

5 Technical specifications

No Technical Specifications are involved in this proposal.

6 Frequently asked questions

7 Acknowledgements

Thanks to Richard Smith for his extensive thoughts on the feasibility, and best formulation, of this proposal.

Thanks to Arthur O'Dwyer for his feedback from his alternative relocatable proposal.

8 References

[N4034] Pablo Halpern,
Destructive Move
https://wg21.link/N4034

[P0023] Denis Bider,
Relocator: Efficiently moving objects
https://wg21.link/P0023

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions
https://wg21.link/P0709

[P0784] Dionne, Smith, Ranns and Vandevoorde,
Standard containers and constexpr
https://wg21.link/P0784