DEEEER0 draft 4: SG14 `[[move_relocates]]`

Document #:	DEEEER0 draft 4
Date:	2018-04-19
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 such attributed types than is possible at present.

The first 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 rather than trivially copyable handles to slots in global memory.

The second major motivation behind this proposal is to broaden the scope of what the compiler's optimiser can treat as copyable and movable without user defined side effects, which we know from trivially copyable type can significant improve the quality and density of codegen in aggregate.

Changes since draft 3:

- Move constructor must now always be defined.
- Base classes and member variables must also have attribute defined.
- Dropped the library stuff entirely as Arthur's paper seems to have that covered.

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

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 when the move of a type with non-trivial move constructor can be substituted with 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:

  - This simple, single purpose, language-only proposal only affects how moves are implemented. It does not change how move works. It does not change object lifetimes.

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

  This proposal differs from relocators in the following ways:

  - 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, attribute opt-in, optimisation of the implementation of move construction only. We do not fiddle with allocators, the meaning nor semantics of moves, object lifetimes, destructors, nor anything else.

All we propose is that where the programmer has indicated that it is safe to do so, we substitute the calling of the move constructor with the fixed operation of `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:
    - Not all base classes are either trivially copyable or `[[move_relocates]]`.
    - If there is a virtual inheritance anywhere in the inheritance tree.
    - Not all member data types are either trivially copyable or `[[move_relocates]]`.
    - The type does not have a public, non-deleted, constexpr, in-class defined default constructor.
    - The type does not have a public, non-deleted, move constructor.
3. If a type `T` has non-ignored attribute `[[move_relocates]]`, the compiler will substitute the move constructor with an as-if `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 implemented 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. It is nice if you add an informative comment mentioning the `[[move_relocates]]`, as the move constructor is never called on types with non ignored `[[move_relocates]]`.
5. If a type `T` has non-ignored attribute `[[move_relocates]]`, the trait `std::is_move_relocating<T>` shall be true.

3.1 Worked example, and effect on codegen

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;

  constexpr unique_ptr(unique_ptr &&o) noexcept : _v(o._v)  // [[move_relocates]]
  {
    o._v = nullptr;
  }
  unique_ptr &operator=(unique_ptr &&o) noexcept
  {
    delete _v;
```

```
    _v = o._v;
    o._v = nullptr;
    return *this;
  }
  ~unique_ptr()
  {
    delete _v;
    _v = nullptr;
  }

  T &operator*() noexcept { return *_v; }
};
```

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, will be reduced by the optimiser to 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:

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

extern unique_ptr<int> __attribute__((noinline)) foo()
{
  auto a = boo();
  *a += *boo();
  return a;
}

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

3.1.1 With current compilers:

On current C++ compilers[1], the program will generate the following x64 assembler:

```
boo():
  push rbx
  mov rbx, rdi
  mov edi, 4
  call operator new(unsigned long)
```

[1] GCC 8 trunk as of a few days ago with `-O2` on.

```
  mov QWORD PTR [rbx], rax
  mov rax, rbx
  pop rbx
  ret
```

As unique ptr is not a trivially copyable type, the compiler is forced to use stack storage to return the unique ptr. The caller passes in where it wants the return stored in `rdi`, which is saved into `rbx`. It allocates four bytes (`edi`) for the int using operator new, and places the pointer to the allocated memory into the eight bytes pointed to by `rbx`. It returns the pointer to the pointer to the allocated int via `rax`.

```
foo():
  push rbp
  push rbx
  mov rbx, rdi
  sub rsp, 24
  call boo()
  lea rdi, [rsp+8]
  call boo()
  mov rdi, QWORD PTR [rsp+8]
  mov rax, QWORD PTR [rbx]
  mov esi, 4
  mov edx, DWORD PTR [rdi]
  add DWORD PTR [rax], edx
  call operator delete(void*, unsigned long)
  add rsp, 24
  mov rax, rbx
  pop rbx
  pop rbp
  ret
  mov rbp, rax
  jmp .L5
foo() [clone .cold.1]:
.L5:
  mov rdi, QWORD PTR [rbx]
  mov esi, 4
  call operator delete(void*, unsigned long)
  mov rdi, rbp
  call _Unwind_Resume
```

We firstly allocate 24 bytes on the stack frame (`rsp`) for the two unique ptrs, calling `boo()` twice to fill each in. We load the two pointers to the two `int`'s from the two unique ptrs (`rdi`, `rax`), dereference that into the allocated `int` for one (`edx`) and add it directly to the memory pointed to by `rax`. We call operator delete on the added-from unique ptr, returning the added-to unique ptr.

```
main:
  sub rsp, 24
  lea rdi, [rsp+8]
  call foo()
  mov rdi, QWORD PTR [rsp+8]
  mov esi, 4
  call operator delete(void*, unsigned long)
  xor eax, eax
  add rsp, 24
  ret
```

After reserving space for the returned unique ptr filled in by calling `foo()`, `main()` loads the pointer to the allocated memory returned by `foo()`, and calls operator delete on it. This is unique ptr's destructor correctly firing on destruction of the unique ptr.

3.1.2 With the proposed `[[move_relocates]]`:

Now let us look at the x64 assembler which would be generated instead if this proposal were in place:

```
boo():
  mov edi, 4
  jmp operator new(unsigned long) # TAILCALL
```

The compiler now knows that unique ptrs can be stored in registers because moves relocate. Knowing this, it optimises out entirely the use of stack to transfer instances of unique ptrs, and thus simply returns in `rax` a naked pointer to a four byte allocation for the `int`. In other words, the `unique_ptr` implementation is entirely eliminated, just its data member an `int*` remains!

```
foo():
  push rbx
  call boo()
  mov rbx, rax
  call boo()
  mov esi, 4
  mov edx, DWORD PTR [rax]
  add DWORD PTR [rbx], edx
  mov rdi, rax
  call operator delete(void*, unsigned long)
  mov rax, rbx
  pop rbx
  ret
```

`foo()` has become rather simpler, too. `boo()` returns the allocated int directly in `rax`, so now the compiler can simply dereference one of them once, add it to the memory pointed to by the other. No more double dereferencing!

The first unique ptr is destructed, and we return the second unique ptr in `rax`.

```
main:
  call foo()
  mov esi, 4
  mov rdi, rax
  call operator delete(void*, unsigned long)
  xor eax, eax
  ret
```

`main()` has become almost trivially simple. We call `foo()`, and delete the pointer it returns before returning zero from `main()`.

3.2 So what?

Those of you who are used to counting assembler opcode latency will immediately see that the second edition is many times faster than the first edition *because it depends on memory much less*. Even though reads and writes to the stack are probably L1 cache fast, any read or write to memory is far slower than CPU registers, typically a maximum of one operation per cycle with a latency of as much as three cycles. CPU registers typically can issue four operations per cycle, with between a zero and one cycle latency. If you add up the CPU cycles in the two examples above, excluding operators new and delete, you will find the second example is several times faster with a fully warmed L1 cache.

What is hard to describe to the uninitiated is how well this microoptimisation aggregates over a whole program. If you make all the types in your program trivially copyable, you will see across the board performance improvements with especial gain in *performance consistency*.

This is why SG14, the low latency study group, would really like for WG21 to standardise relocation so a greater range of types can be brought under maximum optimisation, including [P0709] *Zero-overhead deterministic exceptions* and [PGGGG] *Low level file i/o library*, both of which would make great use of move relocates.

4 Design decisions, guidelines and rationale

Previous work in this area has tended towards the complex. This proposal proposes the barest of essentials for a limited subset of 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.

Thanks to Nicol Bolas for quite extensive feedback and commentary.

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

[PGGGG] Douglas, Niall
Low level file i/o library
https://wg21.link/PGGGG