P1029R3: move `= bitcopies`

Document #:	P1029R3
Date:	2020-01-12
Project:	Programming Language C++ Evolution Working Group
Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

This proposes a new default for C++ move constructors `= bitcopies`, which enables more aggressive optimisation of move constructions for such types than is possible at present.

The primary motivation of this proposal is to propose a form of move relocation so unambitious, uncontentious and conservative that it has a realistic chance of getting approved by WG21 for C++ 23. Indeed, one objection to R2 in Belfast was any use of the word 'relocate', so in R3 we no longer say that word in order to avoid that class of objection as well. We now speak in terms of move bitcopying only.

Something similar in effect, though not in semantics, to this proposed feature is already in the clang compiler via the `[[clang::trivial_abi]]` attribute[1]. The main difference is that this proposal also supports move bitcopying polymorphic types.

Changes since R2:

- We no longer use the word 'relocate' in this proposal, lest there be any concern that the 'empty relocation space' be affected by this proposal.
- A clarifying note was added that in derived classes, defaulting the move constructor does not propagate bitcopying move semantics even if all base classes and member variables have bitcopying move constructors.
- Added Gasper's EWG-I suggestion that `= bitcopies(auto)` ought to be how a derived class defaults the move constructor to bitcopying if all base classes and member variables have bitcopying compatible move constructors.
- Removed all reference to prior papers on relocation, because some felt those citations were staking a claim to 'the relocation space'. Those interested in additional reading should consult the R2 version of this paper.

Contents

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

1 Introduction

The most aggressive optimisations which the C++ compiler can currently 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 at its convenience 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 relocated arbitrarily, at any time and for any reason, in memory as if by `memcpy`. For example, a `std::unique_ptr<T>` implementation might have a similar implementation to:

```
template<class T> class unique_ptr
{
  T *_ptr{nullptr};
public:
  unique_ptr() = default;
  unique_ptr(unique_ptr &&o) : _ptr(o._ptr) { o._ptr = nullptr; }
  ~unique_ptr() { delete _ptr; _ptr = nullptr; }
  ...
};
```

In current compilers, returning from a function a `std::unique_ptr<T>` will have a much heavier ABI overhead over returning a `T*`, because the lack of trivial copyability means that the compiler must use the stack to return unique ptrs, whereas the naked pointer can be directly returned in a CPU register (see worked example in assembler later in this paper).

With `= bitcopies`, a `unique_ptr` implementation might instead be written:

```
template<class T> class unique_ptr
{
  T *_ptr{nullptr};
public:
  constexpr unique_ptr() = default;
  unique_ptr(unique_ptr &&) = bitcopies;          // This type is move bitcopying!
  ~unique_ptr() { delete _ptr; _ptr = nullptr; }
  ...
};
```

... and the compiler would now know that this type can be arbitrarily relocated in memory, with no ill effect, via the following as-if sequence:

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

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

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

When the move constructor `= bitcopies`, the programmer is giving the *explicit guarantee* to the compiler that for this type:

1. Move construction equals two as-if `memcpy()`'s, one from old storage to new, one from a **constexpr**[2] default constructed instance to old.
2. That any non-trivial destruction of a default constructed instance of the type *has no side effects* (and thus can be safely elided by the compiler if it knows that an instance is default constructed).

Because of these warranties made by the programmer to the C++ compiler, returning STL containers by value from functions can now be optimal in terms of codegen (see worked example in assembler later in this paper). A `std::vector<T>` with default allocator might have a similar implementation to:

```
template<class T> class vector
{
  T *_begin{nullptr}, *_end{nullptr}, *_capacity{nullptr};
public:
  constexpr vector() = default;

  vector(vector &&) = bitcopies;
  // memcpy src to dest, then memcpy constexpr default instace over src i.e. same as:
```

[2]In this proposal, one may only use `= bitcopies` with types with a in-class defined, constexpr default constructor.

```
  // vector(vector &&o) : _begin(o._begin), _end(o._end), _capacity(o._capacity) { o._begin = o._end =
       o._capacity = nullptr; }

  // delete of a nullptr has no side effects, so this destructor implementation
  // meets the guarantee given to the compiler.
  ~vector() { delete _begin; _begin = _end = _capacity = nullptr; }
  ...
};
```

Because the compiler knows that this type is move bitcopyable, and destructing moved-from instances has no side effects, it can bit copy the instance into CPU registers for the return rather than using the stack, if the target architecture has sufficient CPU registers return capacity. This brings the same power of optimisation to a large subset of non-trivially-copyable C++ types.

2 Proposed Design

1. That a first new default choice `= bitcopies` become applicable to move constructors. The programmer applies this implementation if they wish to guarantee to the compiler that the move constructor and destructor implementation have stronger guarantees than usual.
2. That a second new default choice `= bitcopies(auto)` become applicable to move constructors. The programmer applies this implementation if they wish the compiler to examine the move constructors of all base types and member variables, and mark this type's move constructor as `= bitcopies` if all base types and member variables have a move constructor with `= bitcopies` compatible semantics.
3. It shall be a compile time diagnostic if:
   - Not all base classes are either trivially copyable, or there is a move constructor in a base class without `= bitcopies` semantics.
   - If there is a virtual inheritance anywhere in the inheritance tree.
   - Not all member variable data types are either trivially copyable, or any member data type has a move constructor without `= bitcopies` semantics.
   - The type does not have a non-deleted, constexpr, in-class defined default constructor. This implies that all base classes and member variables must have a constexpr, in-class defined constructor. Note that the default constructor need not be public, `= bitcopies` is a move constructor implementation, and thus can call non-public member functions.
4. Types with virtual destructors and `= bitcopies` move constructors are permitted. If the programmer destructs a such a type, and the compiler knows it contains a default constructed instance, the compiler may skip calling the virtual destructor.

   [*Note:* This has a mild corner case risk: move constructing a derived object via its base type would reset the vptr of the moved-from object to the vptr of the base type, which some users might find surprising, because any later destruction of the moved-from object will not call the derived destructor implementation. However so long as the destructor of derived classes has no side effects when called on a moved-from instance of itself, the overall outcome remains safe, and within programmer expectations. – end note]

5. If a type `T`'s move constructor has been defaulted to `= bitcopies`, the compiler will implement 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 the second memory copy. The destructor is not ordinarily called on the source object, as the programmer has guaranteed that doing so on a default constructed instance has no side effects.

6. If a type `T`'s move constructor has `= bitcopies` compatible semantics (which includes trivial copyability), the trait `std::is_move_constructor_bitcopying<T>` shall be true.

7. Finally, the C++ standardese for this proposal would guarantee that move bitcopying types can *pass through* code with a C ABI. In other words, it would be defined behaviour for a move bitcopying type to be passed to an `extern "C"` function by value, under C2x's compatible type rules. If the C ABI function is implemented in C, it would bit copy the value, and having no knowledge of destructors, it would not call destructors on copied-from instances. It would be undefined behaviour to send a move bitcopying object into C, and for that object instance to not eventually return to C++.

   This may seem superfluous, but it would be a great boon to aid interoperation with other languages, which speak C. Right now, efficient other-language bindings generally must do lots of UB to avoid excessive memory copying and dynamic memory allocation for type erasure. If C++ could legally send a richer subset of C++ object types to C, particularly the use case of rich C++ types *passing through* C code e.g. via a C callback function, that would be very useful in removing the need for much UB in language interop, and enable a much larger subset of C++ to be directly invocable by C code.

2.1 Worked example, and effect on codegen

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

```
template<class T>
class unique_ptr
{
  T *_v{nullptr};
public:
  // Has an in-class defined, 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;

  unique_ptr(unique_ptr &&) = bitcopies;
  unique_ptr &operator=(unique_ptr &&o) noexcept
  {
    delete _v;
    _v = o._v;
    o._v = nullptr;
    return *this;
  }
  ~unique_ptr()
  {
    delete _v;      // No side effects when _v == nullptr
    _v = nullptr;
  }

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

The default constructor is not deleted, constexpr and defined in-class, and it sets the single, trivially copyable, member data `_v` to `nullptr`. No base classes nor member variables are neither trivially copyable nor move relocating, so the application of `= bitcopies` does not cause a compile time diagnostic.

The destructor, when called on a default constructed instance, will be reduced by the optimiser to a trivial destructor (`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 shall compile this small program and see how it looks before and after the attribute has been applied:

```
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;
}
```

2.1.1 With current compilers, without `= bitcopies`:

On current C++ compilers[3], the program will generate the following x64 assembler:

```
boo():
```

[3]GCC 8 with `-O2` on.

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

2.1.2 With the proposed `= bitcopies`:

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()`.

2.1.3 How do you know that the code in the second example is feasibly generatable by a compiler?

The second example is not hand written. I actually created two unique ptr implementations, one trivially copyable and one the above, and used forced casting to introduce trivially copyable semantics at the correct points. The code you see above was actually generated by a mixture of clang trunk and GCC trunk, using those forced type castings to mimic the proposed semantics.

Upon reviewing this paper, Richard Smith suggested that applying the `[[clang::trivial_abi]]` attribute might result in similar elision of `unique_ptr`. This was tested and found to be true.

2.2 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: Throwing values* and [P1031] *Low level file i/o library*, both of which would make great use of move relocates.

3 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.

4 Technical specifications

No Technical Specifications are involved in this proposal.

5 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, and to Alberto Barbati for feedback helping me reduce the size of the proposal still further.

6 References

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

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

[P1028] Douglas, Niall
SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions
https://wg21.link/P1028

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031