P0889: Ultimate copy elisions
and
P0878R0: Subobjects copy elision

Antony Polukhin

Copy / Move elision

Copy/Move elision

- Simplified description:
  - Copy elision allows to not call copy/move constructors when returning a value of function return type from a function

Copy/Move elision

- Simplified description:
  - Copy elision allows to not call copy/move constructors when returning a value of function return type from a function

- Precise description available at [class.copy.elision]

Some code

```
struct T {
    T() noexcept;  T(T&&) noexcept; ~T() noexcept;
    void do_something() noexcept;
};

static T produce()   { T a; a.do_something(); return a; }
static T update(T b) { b.do_something(); return b; }
static T shrink(T c) { c.do_something(); return c; }

int caller() {
    T d = shrink(update(produce()));
}
```

Current result of compilation

```
caller():
  sub rsp, 24
  lea rdi, [rsp+14]
  call T::T()
  lea rdi, [rsp+14]
  call T::do_something()
  lea rdi, [rsp+14]
  call T::do_something()
  lea rsi, [rsp+14]
  lea rdi, [rsp+15]
  call T::T(T&&)
  lea rdi, [rsp+15]
  call T::do_something()
  lea rsi, [rsp+15]
  lea rdi, [rsp+13]
  call T::T(T&&)
  lea rdi, [rsp+15]
  call T::~T()
  lea rdi, [rsp+14]
  call T::~T()
  lea rdi, [rsp+13]
  call T::~T()
```

Current result of compilation

```
caller():
  call T::T()              // T a
  call T::do_something()   // a.do_something();
                           // T& b = a; // copy elision worked well
  call T::do_something()   // b.do_something();
  call T::T(T&&)           // T c{std::move(b)};
  call T::do_something()   // c.do_something();
  call T::T(T&&)           // T d{std::move(c)};
  call T::~T()             // ~d();
  call T::~T()             // ~c();
  call T::~T()             // ~a(); /*~b()*/
```

Current result of compilation

```
caller():
  call T::T()                // T a
  call T::do_something()     // a.do_something();
                             // T& b = a; // copy elision worked well
  call T::do_something()     // b.do_something();
  call T::T(T&&)             // T c{std::move(b)};
  call T::do_something()     // c.do_something();
  call T::T(T&&)             // T d{std::move(c)};
  call T::~T()               // ~d();
  call T::~T()               // ~c(); ← `c` isn't accessed after move construction of `d`
  call T::~T()               // ~a(); ← `a` isn't accessed after move construction of `c`
```

There's something strange

- In assembly there is a following pattern:
  - Variable X is copy constructed from variable Y
  - Variable Y is not accessed any more
  - Variable Y is destroyed

There's something strange

- In assembly there is a following pattern:
  - Variable X is copy constructed from variable Y
  - Variable Y is not accessed any more
  - Variable Y is destroyed

- Instead if copying and using the copy compiler could reuse the old object as if it was a new one (do the copy elision) [*]

Better result is possible:

```
caller():
  call T::T()               // T a
  call T::do_something()    // a.do_something();
  call T::do_something()    // ba.do_something();
  call T::T(T&&)            // T c{std::move(b)};
  call T::do_something()    // ca.do_something();
  call T::T(T&&)            // T d{std::move(c)};
  call T::~T()              // ~d();
  call T::~T()              // ~c();
  call T::~T()              // ~a();
```

Better result is almost 2 times shorter:

```
caller():
  call T::T()                 // T a
  call T::do_something()   // a.do_something();
  call T::do_something()   // a.do_something();
  call T::do_something()   // a.do_something();
  call T::~T()                // ~a();
```

The idea: Relaxed rules for CE

- Allow to reuse the old object as if it was a new one if the old object is not accessed between a copy/move construction of it and its destruction.
- Allow modern compilers to mix copy elision optimization with results of other optimizations.

Problems that P0889 addresses

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times
- Compilers fail to inline the copy/move constructors and optimize the whole code to the same point as the relaxed copy elision rules may allow

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times
- Compilers fail to inline the copy/move constructors and optimize the whole code to the same point as the relaxed copy elision rules may allow:
  - Affects runtime performance
  - Affects binary sizes

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times
- Compilers fail to inline the copy/move constructors and optimize the whole code to the same point as the relaxed copy elision rules may allow:
  - Affects runtime performance
  - Affects binary sizes
- Because compilers do not elide copy constructors users are forced to write std::move

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times
- Compilers fail to inline the copy/move constructors and optimize the whole code to the same point as the relaxed copy elision rules may allow:
  - Affects runtime performance
  - Affects binary sizes
- Because compilers do not elide copy constructors users are forced to write std::move
  - Rules for writing (or not writing) std::move are tricky

Problems that P0889 addresses

- Compilers are forced to inline constructors+destructors and optimize a lot of IR
  - Affects compile times
- Compilers fail to inline the copy/move constructors and optimize the whole code to the same point as the relaxed copy elision rules may allow:
  - Affects runtime performance
  - Affects binary sizes
- Because compilers do not elide copy constructors users are forced to write std::move
  - Rules for writing (or not writing) std::move are tricky
    - Affects language usage simplicity and teach-ability

How far we should go and what could be improved?

Copy elisions through references

- return std::move(local_variable);
- auto& v = local_variable; return v;
- return path(__lhs) /= __rhs; // libstdc++/85671
- ???

Decompose default destructible types

- return pair.second;
- return get<0>(tuple); // requires CE through references
- auto [a,b] = foo(); return a; // requires CE through references
- return local_aggregate_variable.name;

Decompose any type

```
return stringstream.str();        // requires CE through references
return get<int>(variant);         // requires CE through references
return path.string();
```

Not only for function returns

- `{ T v;   takes_by_copy(v); }`
- `{ T v;   takes_by_reference_and_copies_internally(v); }`
- `struct B { T a; B(const T& a): a(a) {} };       B b{T{}};              // CWG #1049`

All together (P0889)

[class.copy.elision]

...

Additionally, copy elision is allowed for any non-volatile object with automatic storage duration and it's non-volatile nested objects if source is not accessed between a copy/move construction of it and its destruction.

How far we should go?

- Allow to do copy elisions through references?
- Allow to decompose default destructible types and do copy elisions for subobjects? (P0878R0 “V. Proposed wording 1”)
- Allow to decompose and do copy elisions for subobjects? (P0878R0 “V. Proposed wording 2”)
- Allow elisions to work not only for functions' returns?

- Allow all of the above? (P0889R0 “VI. Proposed wording”)

FAQ

[*] Does it break user code?

[*] Does it break user code?

> Yes

[*] Does it break user code?

> Yes, I'm 100% sure

[*] Does it break user code?

- Yes, I'm 100% sure
- Good news: it breaks only **unportable** code
  - Code where generally-accepted constraint for a copy constructor is not satisfied: "After the definition T u = v;, u is equal to v".

Why the code is unportable?

- Language and Library heavily rely on “**After the definition T u = v;, u is equal to v**”
  - Ranges TS force that requirement
  - Library implicitly requires objects after copy/assignment/move to be equal in [container.requirements.general]
  - Algorithms do not work well if that constraint is not satisfied
    - Complexities are described as "At most [...] swaps" or "Approximately [...] swaps"
    - Algorithms sometimes do not specify the order of copying/swapping
  - [class.copy.elision] implicitly relies on that constraint
    - [class.copy.elision] is not mandatory!
  - Guaranteed copy elision implicitly relied on that constraint

Why the code is unportable?

- C++ Language and Library heavily rely on “After the definition T u = v;, u is equal to v”

- WG21 has been relying on that constraint for a long time and classes that violate that constraint are already unportable across platforms/Standard versions.

Is it important?

Are those problems important for users?

- Runtime performance
- Binary sizes
- Compile times
- Language usage simplicity and teach-ability

Those problems are important for users!

- Runtime performance
- Binary sizes
- Compile times
- Language usage simplicity and teach-ability

- Here are only some EWG papers from 2018 mailings that are related to those problems:
  - [[move_relocates]], [[likely]], trivial virtual destructors, zero-overhead exceptions, [[no_unique_address]], Modules, down with typename ...
- There's even more papers for LEWG that try to improve some of those

Is it possible to implement right now?

Is it possible to implement right now?

Yes, but it would be hard.

Anyway, this proposal does not require any of the optimizations from examples. The proposal simply attempts to relax copy elision rules to allow those optimizations someday.