P0889
Ultimate copy elisions

Полухин Антон

Antony Polukhin

Yandex Taxi

Motivation

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()                // ~c();
  call T::~T()                // ~a(); /*~b()*/
  call T::~T()                // ~d();
```

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()                // ~c(); ← `c` isn't accessed after move construction of `d`
  call T::~T()                // ~a(); ← `a` isn't accessed after move construction of `c`
  call T::~T()                // ~d();
```

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()                // ~c();
  call T::~T()                // ~a();
  call T::~T()                // ~d();
```

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.

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()                // ~c();
  call T::~T()                // ~a(); /*~b()*/
  call T::~T()                // ~d();
```

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 and stack usage
  - 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 and stack usage
  - 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 and stack usage
  - 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 and stack usage
  - 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

More examples?

Copy elisions through references

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

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
- http_builder()->get("example.org")->args("foo=bar")->run();
- ???

How far we should go?

- Copy elision is allowed for any object with automatic storage duration if source is not accessed between a copy/move construction of it and its destruction.
- Copy elision is allowed for any **non-volatile** object with automatic storage duration if source is not accessed between a copy/move construction of it and its destruction.
- Copy elision is allowed for any non-volatile object with automatic storage duration if source is not accessed between a copy/move construction of it and its destruction **and source outlives target**.
- Copy elision is allowed for any non-volatile object with automatic storage duration if source is not accessed between a copy/move construction of it and its destruction and source **is destroyed immediately** after target.

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.

Спасибо!

Thanks for listening!

Antony Polukhin

Developer in Yandex.Taxi

antoshkka@gmail.com

antoshkka@yandex-team.ru

https://github.com/apolukhin

C++ РГ21 C++ РОССИЯ https://stdcpp.ru/