Limitations and Problems in std::function and Similar

Mitigations and Alternatives

AMANDEEP CHAWLA

Cppcon
The C++ Conference
2024
September 15 - 20

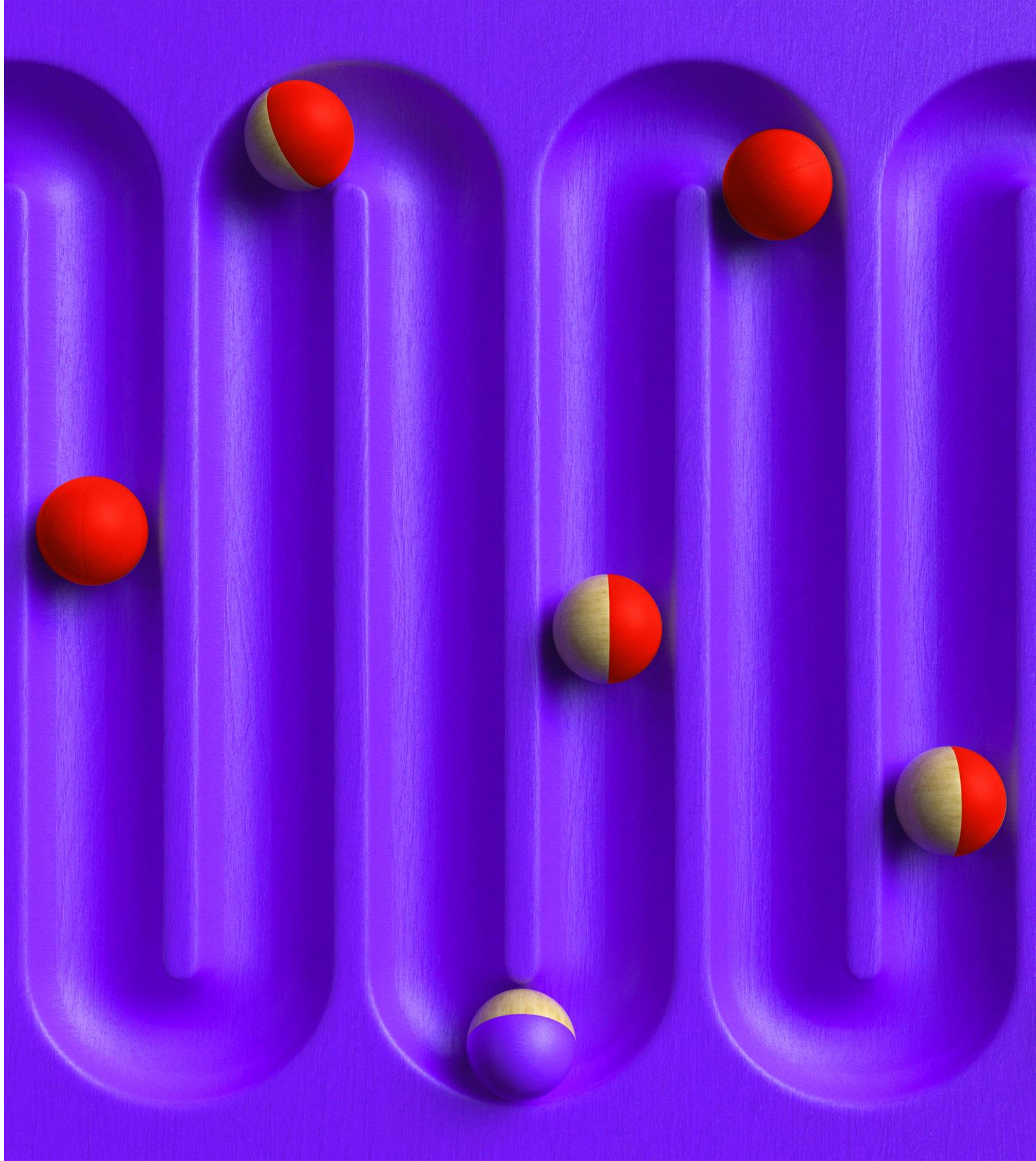
Limitations and Problems in std::function and similar constructs

Amandeep Chawla | Sr. Computer Scientist II

amandeep@adobe.com | adchawla@gmail.com

Adobe

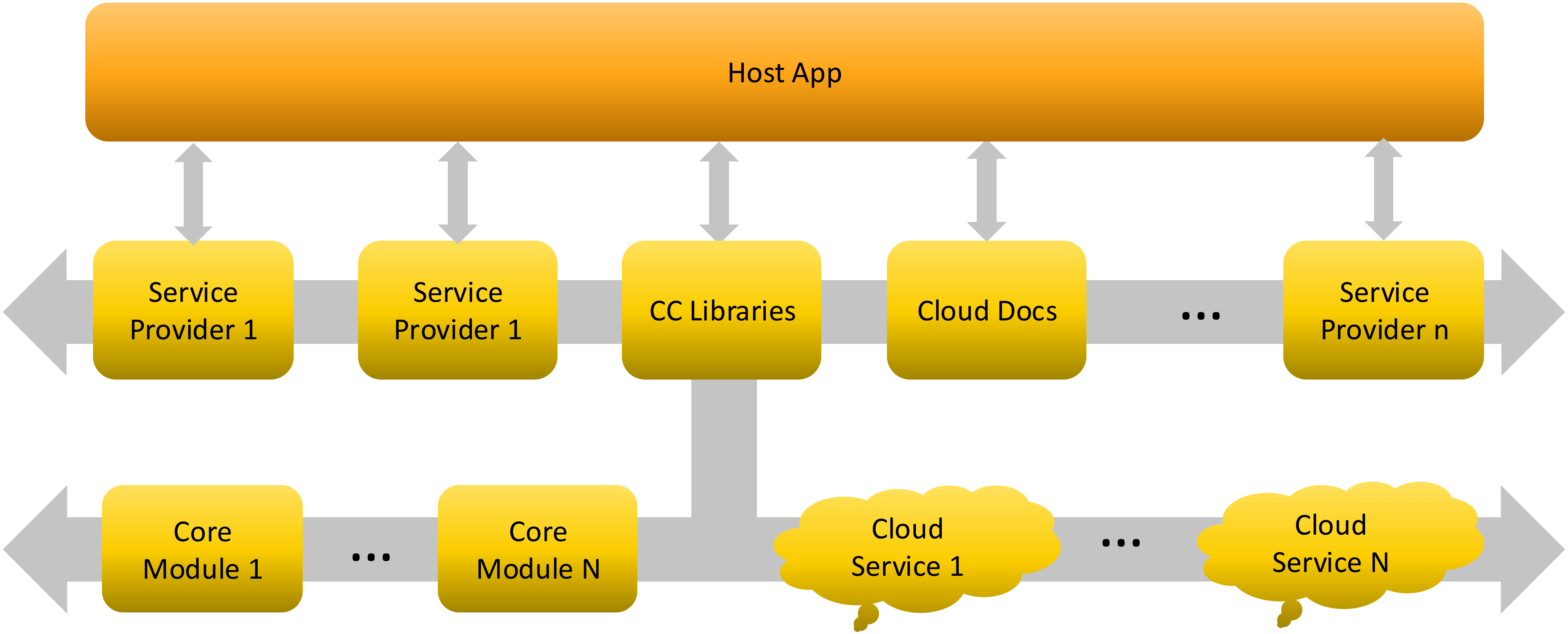
Image by Bruno Tornielli



⚠️ Important: Code is Slide-Aware! ⚠️

- The code snippets in this presentation are designed to fit the slides.
- In some cases, this means:
    - **Simplified examples** for clarity and space.
    - **Omitted error handling** or **non-essential details**.
    - **Lines may be wrapped** to fit the format.
- **Takeaways**:
    - Focus on concepts and patterns
- **Stay engaged and ask questions** if anything is unclear!

10000 ft view



Host Apps

Desktop (Windows/Mac)

- Adobe Photoshop
- Adobe Illustrator
- Adobe InDesign
- Adobe Fresco
- Adobe XD
- Adobe Substance 3D
- Adobe Premiere Pro
- Adobe Premiere Pro
- Adobe Creative Cloud

Mobile

- Adobe Photoshop
- Adobe Illustrator
- Adobe Fresco
- Adobe Capture
- Adobe Premiere Rush
- Adobe Aero
- Adobe Creative Cloud

Web

- Adobe Photoshop
- Adobe Express
- Adobe Stock
- Adobe Fonts
- Adobe Creative Cloud

Initialization Phase

```
void initialize(
  ConfigurationSettings settings,
  TaskQueuePtr syncTQ, TaskQueuePtr bgTQ, TaskQueuePtr notifyTQ,
  std::function<void(ValueOrError<bool>)> callback
);
```

Adobe

Lambdas

```
// print all the elements of vector of int.
std::for_each(c.begin(), c.end(), [](int i) {
    std::cout << i << ' ';
});

// Sort the vector<int> using a lambda function
std::sort(v.begin(), v.end(), [](int a, int b) {
    return a < b;
});

// find the object with id "c" in a vector of objects
// having an id() function returning string
const auto it = std::find_if(v.begin(), v.end(), [&](const auto & obj) {
    return obj.id() == "c";
});
```

InstrumentedClass

```
struct InstrumentedClass {
    explicit InstrumentedClass(std::string id) : id_(move(id)) {}
    ~InstrumentedClass() = default;

    // id_ = "C(" + id_ + ")"
    InstrumentedClass(const InstrumentedClass & other);
    // id_ = "M(" + id_ + ")"
    InstrumentedClass(InstrumentedClass && other) noexcept;
    // id_ = "c=(" + id_ + ")"
    InstrumentedClass & operator=(const InstrumentedClass & other);
    // id_ = "m=(" + id_ + ")"
    InstrumentedClass & operator=(InstrumentedClass && other) noexcept;

    const std::string & id() const noexcept { return id_; }
protected:
    std::string id_;
};
```

Quiz: Lambda Captures (by Reference)

```
// assume it is a 64-bit machine and sizeof(InstrumentedClass) == 32
InstrumentedClass obj1{"a"};
InstrumentedClass obj2{"b"};

auto lambda = [&] {
    return std::make_tuple(obj1.id(), obj2.id());
};

auto [id1, id2] = lambda();

OSTREAM << "(" << id1 << ", " << id2 << ")"
OSTREAM << "sizeof(lambda) = " << sizeof(la
```

```
(a, b)
sizeof(lambda) = 16
```

Adobe

Quiz: Lambda Captures (by Value)

```cpp
// assume it is a 64-bit machine and sizeof(InstrumentedClass) == 32
InstrumentedClass obj1{"a"};
InstrumentedClass obj2{"b"};

auto lambda = [=] {
    return std::make_tuple(obj1.id(), obj2.id());
};

auto [id1, id2] = lambda();

OSTREAM << "(" << id1 << ", " << id2 << ")"
OSTREAM << "sizeof(lambda) = " << sizeof(la
```

```
(C(a), C(b))
sizeof(lambda) = 64
```

Adobe

"Lambda expressions are anonymous functions that can be defined at the point where they are used. They make your code more concise and readable."

- C++ Programming Language, 4th Edition, Bjarne Stroustrup

Lambdas are so cool, but...

- Can they be
  - Used as data member of a class?
  - Stored in containers like queue/vector?
- No, because
  - They return a Closure, and the Closure type is unique unnamed class type.
  - Size is different based on the captures
- So, lambdas can't be used directly in Task based frameworks as we need something whose type we can control.
  - std::function

Adobe

Task Based Mechanism

```cpp
class TaskQueue {
public:
    using Task = std::function<void()>;

    TaskQueue();
    ~TaskQueue();

    void enqueue(const Task & task);
    void shutdown();
};
```

Adobe

std::function

- Is a class template
- It is a general-purpose polymorphic function wrapper
- Instances of std::function can store, copy, and invoke any ***CopyConstructible Callable*** target
- Uses type-erasure under the hood to gain all the magical powers
- Utilizes small-size optimization in case target size is within certain limits

Our Goals

- Is to schedule function on the task queue by moving all the relevant parameters.
- Once the function is executed, call the provided callback to notify the user with the results

```
void asyncFn(
    InstrumentedClass byValue,
    InstrumentedClass & byRef,
    const InstrumentedClass & byCRef,
    const std::function<void(std::vector<std::string>)> & callbackFn)
{
    callbackFn(std::vector{ byValue.id(), byRef.id(), byCRef.id() });
}
```

Adobe

QUIZ: Guess the output

```
InstrumentedClass byValue("byValue");
InstrumentedClass byRef("byRef");
InstrumentedClass byCRef("byCRef");
InstrumentedClass capturedInCb("capturedInCb");

auto cbLambda = [capturedInCb = move(capturedInCb)](const auto & ids) {
    for (const auto & id : ids) { OSTREAM << id << ", "; }
    OSTREAM << capturedInCb.id() << endl;
};

taskQueue.enqueue([
    byValue = move(byValue), byRef = move(byRef),
    byCRef = move(byCRef), callbackFn = move(cbLambda)]() mutable {
        asyncFn(byValue, byRef, byCRef, callbackFn);
    }
);
```

Results

Windows – Visual Studio 2022

```
C(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
C(C(M(M(M(capturedInCb)))))
```

Mac OSX – Xcode 15.4

```
C(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
M(C(C(M(M(M(capturedInCb))))))
```

Removing unnecessary copies

```
InstrumentedClass byValue("byValue");
InstrumentedClass byRef("byRef");
InstrumentedClass byCRef("byCRef");
InstrumentedClass capturedInCb("capturedInCb");

auto cbLambda = [capturedInCb = move(capturedInCb)](const auto & ids) {
    for (const auto & id : ids) { OSTREAM << id << ", "; }
    OSTREAM << capturedInCb.id() << endl;
};

taskQueue.enqueue([
    byValue = move(byValue), byRef = move(byRef),
    byCRef = move(byCRef), callbackFn = move(cbLambda)]() mutable {
        asyncFn(move(byValue), byRef, byCRef, move(callbackFn));
    }
);
```

Adobe

Removing unnecessary copies …

Windows – Visual Studio 2022

```
M(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
M(C(M(M(M(capturedInCb)))))
```

Mac OSX – Xcode 15.4

```
M(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
M(M(C(M(M(M(capturedInCb))))))
```

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

❌ There is no need for copies to be made of our data, and we must get rid off.

❌ There are so many move operations happening. Can we define the optimum number and achieve them?

Adobe

Compare other constructs

```
void fn(
  InstrumentedClass byValue,
  InstrumentedClass & byRef,
  const InstrumentedClass & byCRef)
{
  OSTREAM << byValue.id() << ", " << byRef.id() << ", " << byCRef.id() << endl;
}

void fn2(
  InstrumentedClass byValue,
  const InstrumentedClass & byCRef)
{
  OSTREAM << byValue.id() << ", " << byCRef.id() << endl;
}
```

Lambda vs std::bind

```
[
    byValue = move(byValue),
    byRef= std::move(byRef),
    byCRef = move(byCRef)
] () mutable  {
    fn(move(byValue), byRef, byCRef);
}();
```

```
// bind can't directly work with
function having reference parameters
bind(
  fn2,
  move(byValue),
  move(byCRef)
)();
```

```
M(M(byValue))
M(byRef)
M(byCRef)
```

```
C(M(byValue))

M(byCRef)
```

Adobe

std::async

```
async(
  launch::async, fn,
  move(byValue),
  move(byRef),
  move(byCRef)
).get();
```

Windows – Visual Studio 2022

```
M(M(M(byValue)))
M(M(byRef))
M(M(byCRef))
```

```
async(
  launch::async, fn2,
  move(byValue),
  // move(byRef),
  move(byCRef)
).get();
```

Mac OSX – Xcode 15.4

```
M(M(M(M(byValue))))

M(M(M(byCRef)))
```

Summary of comparisons

- Lambdas provide us the maximum flexibility and performs quite well.
- std::async can execute our code on a separate thread
  - No copies involved
  - Implementation on windows gave us lower number of moves
  - but we don’t have much control on the execution context

Setting our Gold Standard

```
InstrumentedClass byValue("byValue");
InstrumentedClass byRef("byRef");
InstrumentedClass byCRef("byCRef");
InstrumentedClass capturedInCb("capturedInCb");

auto cbLambda = [capturedInCb = move(capturedInCb)](const auto & ids) {
    for (const auto & id : ids) { OSTREAM << id << ", "; }
    OSTREAM << capturedInCb.id() << endl;
};

std::async(
  launch::async, asyncFn,
  move(byValue), move(byRef), move(byCRef), move(cbLambda)
);
```

Adobe

Results of our Gold Standard

```
M(M(M(byValue)))
M(M(byRef))
M(M(byCRef))
M(M(M(M(capturedInCb))))
```

Adobe

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

❌ There is no need for copies to be made of our data, and we must get rid off.

✅ There are so many move operations happening. we have defined the optimum number

❌ We haven’t achieved the optimum number of moves with our solution?

Capturing Data in a container

```
struct Holder {
    InstrumentedClass byValue;
    InstrumentedClass byRef;
    InstrumentedClass byCRef;
    CallbackFn callbackFn;

    Holder(InstrumentedClass && byValue, InstrumentedClass && byRef,
           InstrumentedClass && byCRef, CallbackFn && callbackFn)
        : byValue(move(byValue)), byRef(move(byRef))
        , byCRef(move(byCRef)), callbackFn(std::move(callbackFn)) { }
};

auto holder = make_shared<Holder>(move(byValue), move(byRef),
                                  move(byCRef), move(cbLambda));
taskQueue.enqueue([holder = move(holder)] {
    asyncFn(move(holder->byValue), holder->byRef,
            holder->byCRef, move(holder->callbackFn));
});
```

Results vs Old vs Gold Standard

```
M(M(byValue))
M(byRef)
M(byCRef)
M(M(M(capturedInCb)))
```

```
C(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
C(C(M(M(M(capturedInCb)))))
```

```
C(C(M(M(byValue))))
C(M(M(byRef)))
C(M(M(byCRef)))
M(C(C(M(M(M(capturedInCb))))))
```

```
M(M(M(byValue)))
M(M(byRef))
M(M(byCRef))
M(M(M(M(capturedInCb))))
```

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

✅ There is no need for copies to be made of our data, and we must get rid off.

✅ We have defined the optimum number of moves and **bettered** them with our solution.

But

❌ std::function is forcing us to use shared_ptr in place of unique_ptr.

❌ The code is not generic enough

Adobe

MoveWrapper

```
template <typename T>
struct MoveWrapper {
    explicit MoveWrapper(T && value) noexcept : value_{move(value)} { }

    // copy acts like move
    MoveWrapper(const MoveWrapper & src) noexcept : value_{move(src.value_)} { }
    MoveWrapper(MoveWrapper && src) noexcept : value_{move(src.value_)} { }
    MoveWrapper & operator=(const MoveWrapper &) = delete;
    MoveWrapper & operator=(MoveWrapper &&) noexcept = delete;
    ~MoveWrapper() = default;

    T & value() & noexcept { return value_; }
    const T & value() const & noexcept { return value_; }
    T && value() && noexcept { return std::move(value_); }
private:
    mutable T value_{};
};
```

Using MoveWrapper

```
auto holderWrapper = MoveWrapper(std::make_unique<Holder>(
    std::move(byValue), std::move(byRef),
    std::move(byCRef), std::move(cbLambda)));

taskQueue.enqueue([holderWrapper = std::move(holderWrapper)] {
    auto & holder = holderWrapper.value();
    asyncFn(std::move(holder->byValue), holder->byRef,
            holder->byCRef, std::move(holder->callbackFn));
});
```

Adobe

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

✅ There is no need for copies to be made of our data, and we must get rid off.

✅ We have defined the optimum number of moves and **bettered** them with our solution.

✅ We are no longer forced to use shared_ptr.

But

❌ The code is not generic enough

Adobe

STL to our rescue

```
template<typename... Args>
auto make_unique_tuple(Args &&... args) {
    using TupleType = tuple<decay_t<Args>...>;
    return make_unique<TupleType>(forward<Args>(args)...);
}

auto t = make_unique_tuple(move(byValue), move(byCRef), move(cbLambda));

taskQueue.enqueue([mt = MoveWrapper(move(t))] {
    apply(asyncFn2, move(*mt.value()));
});
```

Generic Holder

```cpp
template <typename... Args>
struct Holder {
    tuple<decay_t<Args>...> args;

    template <typename... TArgs>
    explicit Holder(TArgs &&... args) : args(forward<TArgs>(args)...) {
    }

    template <typename Callable>
    void invoke(Callable && fn) { apply(fn, move(args)); }
};

template<typename... Args>
unique_ptr<Holder<Args...>> make_unique_holder(Args &&... args) {
    return make_unique<Holder<Args...>>(forward<Args>(args)...);
}
```

Adobe

Using Generic Holder

```
auto uPtr = make_unique_holder(std::move(byValue), std::move(byCRef),
    std::move(cbLambda));


taskQueue.enqueue([holderWrapper = MoveWrapper(std::move(uPtr)] {
    holder.value()->invoke(asyncFn2);


});
```

Adobe

Summary so far…

- ✅ We have established that std::function<void()> is the signature for each task unit
- ✅ There is no need for copies to be made of our data, and we must get rid off.
- ✅ We have defined the optimum number of moves and **bettered** them with our solution.
- ✅ We are no longer forced to use shared_ptr.
- ✅ The code is generic enough

But

- ❌ Doesn’t support functions with non-const references (even on windows)

Why?

```
using HaveType = tuple<InstrumentedClass, InstrumentedClass,
    InstrumentedClass, CallbackFn>;
using WantType = tuple<InstrumentedClass, InstrumentedClass &,
    const InstrumentedClass &, CallbackFn>;

// We have HaveType but std::apply needs WantType
```

Manual Solution

```cpp
using WantType = std::tuple<InstrumentedClass &&, InstrumentedClass &,
    const InstrumentedClass &, CallbackFn>;

taskQueue.enqueue([holder = MoveWrapper(std::move(uPtr))] {
    auto & tuple = holder.value()->args;
    std::apply(asyncFn, WantType{
        std::move(std::get<0>(tuple)), std::get<1>(tuple),
        std::get<2>(tuple), std::move(std::get<3>(tuple))}
    );
});
```

Adobe

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

✅ There is no need for copies to be made of our data, and we must get rid off.

✅ We have defined the optimum number of moves and **bettered** them with our solution.

✅ We are no longer forced to use shared_ptr.

✅ The code is generic enough

✅ Support functions with non-const references (both platforms)

But

❌ Lot of boiler plate code

Adobe

Automated Solution

- Requires
  - Exact types of the parameters of the Callable
  - Converting tuple of values to tuple of types required by Callable

Converting tuple of values to tuple of References

- Logic
  - Parameter type is **by value** -> **r-value** reference
  - Parameter type is **by reference** -> **l-value** reference

TupleConvertor

```
template <bool is_reference, typename T>
auto conditional_move(T & t) -> std::conditional_t<is_reference, T &, T &&> {
  if constexpr (is_reference) { return t; }
  else { return static_cast<T &&>(t); }
}
template <class T, class SrcTuple, std::size_t... I>
auto make_from_tuple_impl(SrcTuple & src, std::index_sequence<I...>) {
  return std::forward_as_tuple(
    conditional_move<is_reference_v<tuple_element_t<I, T>>>(get<I>(src))...);
}
template <typename... T1> struct TupleConvertor {
  explicit TupleConvertor(std::tuple<T1...> & src) : src_(src) { }
  template <typename DestTupleType> auto convert() {
    return make_from_tuple_impl<DestTupleType>(src_,
      make_index_sequence<tuple_size_v<remove_reference_t<DestTupleType>>>{});
  }
  std::tuple<T1...> & src_;
};
```

Adobe

Using TupleConvertor

```
using WantType = std::tuple<InstrumentedClass &&, InstrumentedClass &,
    const InstrumentedClass &, CallbackFn>;

taskQueue.enqueue([holder = MoveWrapper(std::move(uPtr))] {
    auto & tuple = holder.value()->args;
    std::apply(asyncFn, TupleConvertor(tuple).convert<WantType>());



});
```

Adobe

Summary so far…

✅ We have established that std::function<void()> is the signature for each task unit

✅ There is no need for copies to be made of our data, and we must get rid off.

✅ We have defined the optimum number of moves and **bettered** them with our solution.

✅ We are no longer forced to use shared_ptr.

✅ The code is generic enough

✅ Support functions with non-const references (both platforms)

But

❌ Some boiler plate code

function_traits

```
struct function_traits final {};

template <typename R, typename... Args>
struct function_traits<R(Args...)> final {
    using args_tuple_type = std::tuple<Args...>;
};

template <typename... Args>
using function_traits_args_tuple_t =
    typename function_traits<Args...>::args_tuple_type;

using FnType = function_traits_args_tuple_t<decltype(asyncFn)>;
using ExpectedType = std::tuple<InstrumentedClass,
    InstrumentedClass &, const InstrumentedClass &, const CallbackFn &>;
static_assert(std::is_same_v<FnType, ExpectedType>);
```

Adobe

Plugging-in function_traits

```
using WantType = function_traits_args_tuple_t<decltype(asyncFn)>;


taskQueue.enqueue([holder = MoveWrapper(std::move(uPtr))] {
    auto & tuple = holder.value()->args;
    std::apply(asyncFn, TupleConvertor(tuple).convert<WantType>());



});
```

Adobe

Summary so far...

- ✅ We have established that std::function<void()> is the signature for each task unit
- ✅ There is no need for copies to be made of our data, and we must get rid off.
- ✅ We have defined the optimum number of moves and **bettered** them with our solution.
- ✅ We are no longer forced to use shared_ptr.
- ✅ The code is generic enough
- ✅ Support functions with non-const references (both platforms)

But

- ❌ Very less boiler plate code

Adobe

Using Generic Holder

```
auto uPtr = make_unique_holder(std::move(byValue), std::move(byCRef),
    std::move(cbLambda));


taskQueue.enqueue([holderWrapper = MoveWrapper(std::move(uPtr)] {
    holder.value()->invoke(asyncFn2);


});
```

Adobe

Enhancing Holder

```
template <typename... Args>
struct Holder {
    // other member functions and data members
    template <typename Callable>
    void invoke(Callable && fn) {
        using DType = function_traits_args_tuple_t<decay_t<Callable>>;
        auto destT = TupleConvertor(args);
        apply(fn, destT.template convert<DType>());
    }
};
```

Using Enhanced Holder

```
taskQueue.enqueue([holder = MoveWrapper(std::move(uPtr))] {
    holder.value()->invokeEx(asyncFn);


});
```

Summary

- ✅ We have established that std::function<void()> is the signature for each task unit
- ✅ There is no need for copies to be made of our data, and we must get rid off.
- ✅ We have defined the optimum number of moves and **bettered** them with our solution.
- ✅ We are no longer forced to use shared_ptr.
- ✅ The code is generic enough
- ✅ Support functions with non-const references (both platforms)
- ✅ No Boiler plate code

??

- What about memory allocation?

FnHolder

```
template <typename FnType, typename... Args>
struct FnHolder {
    struct Pack {
        template <typename TFnType, typename... TArgs>
        explicit Pack(TFnType && fn, TArgs &&... args)
            : fn(std::forward<TFnType>(fn))
            , args(std::forward<TArgs>(args)...) { }

        std::decay_t<FnType> fn;
        function_traits_args_by_value_tuple_t<decay_t<FnType>> args;
    };

    MoveWrapper<std::unique_ptr<Pack>> content;

    template <typename TFnType, typename... TArgs>
    explicit FnHolder(TFnType && fn, TArgs &&... args)
```

Using FnHolder

```
auto fnHolder = make_fnholder(asyncFn,
  std::move(byValue), std::move(byRef),
  std::move(byCRef), std::move(cbLambda));

taskQueue.enqueue([fnHolder = std::move(fnHolder)] {
    fnHolder.invoke();
});
```

Summary

- ✅ We have established that std::function<void()> is the signature for each task unit
- ✅ There is no need for copies to be made of our data, and we must get rid off.
- ✅ We have defined the optimum number of moves and **bettered** them with our solution.
- ✅ We are no longer forced to use shared_ptr.
- ✅ The code is generic enough
- ✅ Support functions with non-const references (both platforms)
- ✅ No Boiler plate code
- ✅ No MovebleWrapper or unique_ptr creation exposed to user

Adobe

std::packaged_task

- I was not able to wrap an instance of std::packaged_task inside a std::function.

C2280 'PackingData_PackagedTask_Test::TestBody::<lambda_6aec4a3bdce20826bce1fe93cc9aad02>::<lambda_6aec4a3bdce20826bce1fe93cc9aad02>(const PackingData_PackagedTask_Test::TestBody::<lambda_6aec4a3bdce20826bce1fe93cc9aad02> &)': attempting to reference a deleted function

Adobe

std::apply

- It let you call a callable by using all the elements from the provided tuple

```cpp
void asyncFn2(
    InstrumentedClass byValue, const InstrumentedClass & byCRef,
    const std::function<void(std::vector<std::string>)> & callbackFn)
{
    callbackFn(std::vector{byValue.id(), byCRef.id()});
}

auto t = std::make_tuple(
  std::move(byValue), std::move(byCRef),
apply(asyncFn2, std::move(t));
```

```
M(M(byValue))

M(byCRef)
M(M(M(capturedInCb)))
```

std::forward_as_tuple

- Constructs a tuple of references to the arguments in **args** suitable for forwarding as an argument to a function. The tuple has **rvalue** reference data members when **rvalues** are used as arguments, and otherwise has **lvalue** reference data members.

```
int i{0};
int y{1};

auto t = std::forward_as_tuple(i, std::move(y), 10);
using ExpectedType = std::tuple<int &, int &&, int &&>;
static_assert(std::is_same_v<ExpectedType, decltype(t)>);
```

Functions taking parameters by non-const references

```
namespace std::filesystem {
  // filesystem operations
  path absolute(const path& p, error_code& ec);
  path canonical(const path& p, error_code& ec);
  bool copy_file(const path& from, const path& to, error_code& ec);
  bool create_directories(const path& p, error_code& ec);
  bool create_directory(const path& p, error_code& ec) noexcept;
  void current_path(const path& p, error_code& ec) noexcept;
  bool exists(const path& p, error_code& ec) noexcept;
  uintmax_t file_size(const path& p, error_code& ec) noexcept;
}
```

Adobe

Functions taking parameters by non-const references

```
namespace std {
  template<class T>
  struct atomic {
    bool compare_exchange_weak(
      T& expected, T desired, memory_order success, memory_order failure) noexcept;

    bool compare_exchange_weak(
      T& expected, T desired, memory_order order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(
      T& expected, T desired, memory_order success, memory_order failure) noexcept;

    bool compare_exchange_strong(
      T& expected, T desired, memory_order order = memory_order_seq_cst) noexcept;
  };
}
```