vince.tourangeau@autodesk.com

access()

```
auto thing = a.pointer()->another_pointer()->thing_i_want();
```

```
assert(a.pointer() && a.pointer()->another_pointer());

auto thing = a.pointer()->another_pointer()->thing_i_want();
```

```
assert(a.pointer() && a.pointer()->another_pointer());

if (auto b = a.pointer())
{
    if (auto c = b->another_pointer())
    {
        auto thing = c->thing_i_want();
auto thing =
    [&]()
    {
        if (auto b = a.pointer())
        {
            if (auto c = b->another_pointer())
            {
                return c->thing_i_want();
            }
        }

        assert(false);
        return Thing{};
    }();
```

```cpp
auto thing =
    [&]()
    {
        if (auto b = a.pointer())
        {
            if (auto c = b->another_pointer())
            {
                return c->thing_i_want();
            }
        }

        assert(false);
        return Thing{};
    }();
```

```
auto thing =
    [&]()
    {
        if (auto b = a.pointer())
        {
            if (auto c = b->another_pointer())
            {
                return c->thing_i_want();
            }
        }

        assert(false);
        return Thing{};
    }();
```

```
auto thing =
    [&]() -> std::optional<Thing>
    {
        if (auto b = a.pointer())
        {
            if (auto c = b->another_pointer())
            {
                return c->thing_i_want();
            }
        }

        assert(false);
        return std::nullopt;
    }();
```

```
assert(a.pointer() && a.pointer()->another_pointer());

auto thing = a.pointer()->another_pointer()->thing_i_want();
```

access

```
access(a
access(a, &A::pointer // -> B*
access(a, &A::pointer, &B::another_pointer // -> C*
access(a, &A::pointer, &B::another_pointer, &C::thing_i_want)
```

```
auto thing = access(a, &A::pointer, &B::another_pointer, &C::thing_i_want)
```

```
if (auto thing = access(a, &A::pointer, &B::another_pointer, &C::thing_i_want))
if (auto thing = access(a, &A::pointer, &B::another_pointer, &C::thing_i_want))
{
    thing->do_your_thing();
```

```
if (auto thing = access(a, &A::pointer, &B::another_pointer, &C::thing_i_want))
{
    thing->do_your_thing();
}
else
{
    assert(false);
}
```

```
if (auto thing = access(a, &A::pointer, &B::another_pointer, &C::thing_i_want))
{
    thing->do_your_thing();
}
else
{
    assert(false);
}
```

```
if (auto thing = access(a, &A::pointer, &B::bad_pointer, &C::thing_i_want))
{
    thing->do_your_thing();
}
else
{
    assert(false);
}
```

```
if (auto thing = access(a, &A::pointer, &B::bad_pointer, &C::thing_i_want))
{
    thing->do_your_thing();
}
else
{
    assert(false);
}
```

```
template <class Object, class FirstMem, class... RestMems>
constexpr
decltype(auto)
access(Object&& obj, FirstMem first_member, RestMems... rest_members);
```

```
template <class T>
struct safe_result
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<std::optional<T>>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<T*>
{
    using type = T*;
};

template <class T>
struct safe_result<T&>
{
    using type = T*;
};
```

```
template <class T>
struct safe_result
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<std::optional<T>>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<T*>
{
    using type = T*;
};

template <class T>
struct safe_result<T&>
{
    using type = T*;
};

template <class T>
struct safe_result<std::shared_ptr<T>>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<std::unique_ptr<T>>
{
    using type = T*;
};
```

```
template <class T>
struct safe_result
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<std::optional<T>>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<const std::optional<T>&>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<T*>
{
    using type = T*;
};

template <class T>
struct safe_result<T&>
{
    using type = T*;
};

template <class T>
struct safe_result<std::shared_ptr<T>>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<const std::shared_ptr<T>&>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<std::unique_ptr<T>>
{
    using type = T*;
};

template <class T>
struct safe_result<const std::unique_ptr<T>&>
{
    using type = T*;
};
```

```
#include <type_traits>
```

```
#include <type_traits>

template <class T> struct is_dereferenceable; // defined elsewhere
```

```
#include <type_traits>

template <class T> struct is_dereferenceable; // defined elsewhere
template <class T> struct is_member_accessible; // um, also defined elsewhere
```

```
#include <type_traits>

template <class T> struct is_dereferenceable; // defined elsewhere
template <class T> struct is_member_accessible; // um, also defined elsewhere

template <class T>
static constexpr auto is_dereferenceable_v = is_dereferenceable<T>::value;

template <class T>
static constexpr auto is_member_accessible_v = is_member_accessible<T>::value;
```

```
#include <type_traits>

template <class T> struct is_dereferenceable; // defined elsewhere
template <class T> struct is_member_accessible; // um, also defined elsewhere

template <class T>
static constexpr auto is_dereferenceable_v = is_dereferenceable<T>::value;

template <class T>
static constexpr auto is_member_accessible_v = is_member_accessible<T>::value;

template <class T>
static constexpr auto is_pointer_like =
    is_dereferenceable_v<T> &&
    is_member_accessible_v<T> &&
    std::is_convertible_v<T, bool>;
```

```
template <class T>
struct safe_result
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<std::optional<T>>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<const std::optional<T>&>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<T*>
{
    using type = T*;
};

template <class T>
struct safe_result<T&>
{
    using type = T*;
};

template <class T>
struct safe_result<std::shared_ptr<T>>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<const std::shared_ptr<T>&>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<std::unique_ptr<T>>
{
    using type = T*;
};

template <class T>
struct safe_result<const std::unique_ptr<T>&>
{
    using type = T*;
};
```

```cpp
template <class T>
struct safe_result
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<std::optional<T>>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<const std::optional<T>&>
{
    using type = std::optional<T>;
};

template <class T>
struct safe_result<T*>
{
    using type = T*;
};

template <class T>
struct safe_result<T&>
{
    using type = T*;
};

template <class T>
struct safe_result<std::shared_ptr<T>>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<const std::shared_ptr<T>&>
{
    using type = std::shared_ptr<T>;
};

template <class T>
struct safe_result<std::unique_ptr<T>>
{
    using type = T*;
};

template <class T>
struct safe_result<const std::unique_ptr<T>&>
{
    using type = T*;
};

template <class T>
struct safe_result<CustomPointer<T>>
{
    using type = CustomPointer<T>;
};

template <class T>
struct safe_result<const CustomPointer<T>&>
{
    using type = CustomPointer<T>;
};
```

```cpp
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

```cpp
// somewhere inside access(obj, first, rest…)

decltype(auto) result = access_first(); // result ~= obj->first()

if constexpr (sizeof…(rest) > 0)
{
    return access_member<Assert>(result, rest...);
}
else
{
    using ResultT = decltype(result);

    if constexpr (is_pointer_like<ResultT>)
    {
        return result;
    }
    else if constexpr (std::is_reference_v<ResultT>)
    {
        return &result;
    }
    else
    {
        return std::optional<ResultT>{ result };
    }
}
```

vincetogo@mac.com