Writing a better std::move

Jonathan Müller — `@foonathan`

std::move

```
template <typename T>
constexpr std::remove_reference_t<T>&& move(T&& obj)
{
    return static_cast<std::remove_reference_t<T>&&>(obj);
}
```

Problem #1: `const`

```cpp
void sink(std::vector<int> input);
```

```cpp
const std::vector<int> vec = compute();
sink(move(vec));
```

Solution #1: Prevent const

```cpp
template <typename T>
constexpr std::remove_reference_t<T>&& move(T&& obj)
{
    static_assert(!std::is_const_v<std::remove_reference_t<T>>);
    return static_cast<std::remove_reference_t<T>&&>(obj);
}
```

Problem #2: Ownership

```cpp
void decorate(std::unique_ptr<Widget>& widget)
{
    auto padded = make_padding(move(widget), config.padding);      // #1
    auto border = make_border(move(padded), config.border_width);  // #2
    widget = move(border);                                         // #3
}
```

Problem #2: Ownership

```
void decorate(std::unique_ptr<Widget>& widget)
{
    auto padded = make_padding(move(widget), config.padding);      // #1
    auto border = make_border(move(padded), config.border_width);  // #2
    widget = move(border);                                         // #3
}
```

If #2 throws, `widget` is already gone.

Problem #2: Ownership

```cpp
void decorate(std::unique_ptr<Widget>& widget)
{
    auto padded = make_padding(move(widget), config.padding);
    try {
        auto border = make_border(move(padded), config.border_width);
        widget = move(border);
    } catch (...) {
        widget = move(padded->child());
        throw;
    }
}
```

Rust

Cannot move out of `&mut T`!

When can we safely move?

```
Widget local;
sink(move(local)); // ok
```

When can we safely move?

```
Widget local;
sink(move(local)); // ok
```

```
void f(Widget&& arg)
{
    sink(move(arg)); // ok
}
```

When can we safely move?

```
Widget local;
sink(move(local)); // ok
```

```
void f(Widget&& arg)
{
    sink(move(arg)); // ok
}
```

```
void f(Widget& arg)
{
    sink(move(arg)); // dangerous
}
```

When can we safely move?

```
Widget local;
sink(move(local)); // ok
```

```
void f(Widget&& arg)
{
    sink(move(arg)); // ok
}
```

```
void f(Widget& arg)
{
    sink(move(arg)); // dangerous
}
```

We can safely move if we have ownership.

Solution #2: Prevent non-owning

```cpp
template <typename DecltypeT, typename T>
constexpr std::remove_reference_t<T>&& move_impl(T&& obj)
{
    static_assert(!std::is_const_v<std::remove_reference_t<T>>);
    static_assert(!std::is_lvalue_reference_v<DecltypeT>);
    return static_cast<std::remove_reference_t<T>&&>(obj);
}
```

think-cell

Solution #2: Prevent non-owning

```cpp
template <typename DecltypeT, typename T>
constexpr std::remove_reference_t<T>&& move_impl(T&& obj)
{
    static_assert(!std::is_const_v<std::remove_reference_t<T>>);
    static_assert(!std::is_lvalue_reference_v<DecltypeT>);
    return static_cast<std::remove_reference_t<T>&&>(obj);
}
```

```cpp
#define move(...) move_impl<decltype(__VA_ARGS__)>(__VA_ARGS__)
```

Solution #2: Prevent non-owning

```
Widget local;
sink(move(local)); // ok, decltype(local) is Widget
```

Solution #2: Prevent non-owning

```
Widget local;
sink(move(local)); // ok, decltype(local) is Widget
```

```
void f(Widget&& arg)
{
    sink(move(arg)); // ok, decltype(arg) is Widget&&
}
```

Solution #2: Prevent non-owning

```
Widget local;
sink(move(local)); // ok, decltype(local) is Widget
```

```
void f(Widget&& arg)
{
    sink(move(arg)); // ok, decltype(arg) is Widget&&
}
```

```
void f(Widget& arg)
{
    sink(move(arg)); // dangerous, decltype(arg) is Widget&
}
```

Problem #3: Member access

```cpp
struct WidgetHolder
{
    Widget widget;
};

void f(WidgetHolder& holder)
{
    sink(move(holder.widget));
}
```

Problem #3: Member access

```cpp
struct WidgetHolder
{
    Widget widget;
};

void f(WidgetHolder& holder)
{
    sink(move(holder.widget));
}
```

`decltype(holder.widget)` is `Widget`!

Guideline

Always write `move(obj).member` instead of `move(obj.member)`.

Guideline

Always write `move(obj).member` instead of `move(obj.member)`.

Also applies to `move(obj.fn())` if `fn()` is properly overloaded for rvalue refs.

Solution #3: Enforcing the guideline

```cpp
template <typename DecltypeT, bool IsIdExpression, typename T>
constexpr std::remove_reference_t<T>&& move_impl(T&& obj)
{
    static_assert(!std::is_const_v<std::remove_reference_t<T>>);
    static_assert(!std::is_lvalue_reference_v<DecltypeT>);
    static_assert(IsIdExpression);
    return static_cast<std::remove_reference_t<T>&&>(obj);
}

#define move(...) \
  move_impl<decltype(__VA_ARGS__), is_id_expression(__VA_ARGS__)> \
    (__VA_ARGS__)
```

Poor man's reflection

```
#define is_id_expression(...) is_id_expression_impl(#__VA_ARGS__)
```

Poor man's reflection

```
#define is_id_expression(...) is_id_expression_impl(#__VA_ARGS__)
```

```
constexpr bool is_id_expression(char const* const expr)
{
  for (auto str = expr; *str; ++str)
    if (!std::isalpha(*str) && !std::isdigit(*str)
        && *str != '_' && *str != ':')
      return false;
  return true;
}
```

What if you want to move out of an lvalue reference?

What if you want to move out of an lvalue reference?

```
template <typename T>
constexpr std::remove_reference_t<T>&& move_always(T&& obj)
{
    static_assert(!std::is_const_v<std::remove_reference_t<T>>);
    return static_cast<std::remove_reference_t<T>&&>(obj);
}
```

Conditional move

```cpp
template <typename Arg>
void f(Arg&& arg)
{
    sink(std::forward<Arg>(arg));
}
```

Conditional move

```
template <typename Arg>
void f(Arg&& arg)
{
    sink(std::forward<Arg>(arg));
}
```

Move if we have ownership:

- local variable: yes
- lvalue reference: no
- rvalue reference: yes

Conditional move

```
template <typename DecltypeT, typename T>
constexpr std::remove_reference_t<T>&& move_if_owned_impl(T&& obj)
{
    // DecltypeT is U -> U&& (move)
    // DecltypeT is U& -> U& (no move)
    // DecltypeT is U&& -> U&& (move)
    return static_cast<DecltypeT&&>(obj);
}

#define move_if_owned(...) \
  move_if_owned_impl<decltype(__VA_ARGS__)>(__VA_ARGS__)
```

Preventing member access

```cpp
template <typename DecltypeT, bool IsIdExpression, typename T>
constexpr std::remove_reference_t<T>&& move_if_owned_impl(T&& obj)
{
    static_assert(IsIdExpression);
    return static_cast<DecltypeT&&>(obj);
}

#define move_if_owned(...) \
  move_if_owned_impl<decltype(__VA_ARGS__), is_id_expression(__VA_ARGS__)> \
    (__VA_ARGS__)
```

Conclusion

`move` owning is moved, non-owning is error, argument must be id expr

Conclusion

`move` owning is moved, non-owning is error, argument must be id expr

`move_if_owned` owning is moved, non-owning is copied, argument must be id expr

Conclusion

`move` owning is moved, non-owning is error, argument must be id expr

`move_if_owned` owning is moved, non-owning is copied, argument must be id expr

`move_always` owning is moved, non-owning is moved, argument can be anything

Conclusion

github.com/think-cell/think-cell-library

We're hiring: think-cell.com/cppcon

@foonathan@fosstodon.org
youtube.com/@foonathan

think-cell