Members through a looking glass

Meeting C++ Online

dr Ivan Čukić

KDAB
ivan.cukic@kdab.com, ivan@cukic.co
About me

- KDAB senior software engineer
  *Software Experts in Qt, C++ and 3D / OpenGL*

- Author of the "Functional Programming in C++" book *available in English, Chinese, Korean, Russian, Polish*

- Trainer / consultant

- KDE developer

- University lecturer
COMPOSITION
Composition

Doug McIlroy and Dennis Ritchie
Composition

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```
Composition

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new “features”.
- Expect the output of every program to become the input to another, as yet unknown, program. Don’t clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don’t insist on interactive input.

...  

---

Doug McIlroy, Bell System Technical Journal, 1978
<table>
<thead>
<tr>
<th>Composition</th>
<th>Members</th>
<th>Unite</th>
<th>Abstraction</th>
<th>The End</th>
</tr>
</thead>
</table>

Software design is about composition.
9 times out of 10, a for-loop should either be the only code in a function, or the only code in the loop should be a function (or both).

```c
int f()
{
    for (...)
    {
        ....
    }
}
```

or

```c
for(x...) 
g(x);
```

avoid code outside the loop and inside the loop in the same function.
write(std::string)  \rightarrow  \cdots
\text{to\_string(something\_t)}  \rightarrow  \text{std::string}

\textbf{write(to\_string(something);}
auto compose(auto f, auto g)
{
    return [f, g](auto &&...args) {
        return f(g(FWD(args)...));
    }
}
MEMBERS
Pointers to member functions

```cpp
write(std::string) -> ...
something_t::to_string() -> std::string
write(&something_t::to_string(something));
```
Pointers to member functions

&something_t::to_string : something_t -> std::string
auto compose(auto f, auto g)
{
    return [f, g](auto &&...args) {
        return std::invoke(f,
            std::invoke(g, FWD(args)...));
    }
}
Pointers to member functions

```cpp
accumulate(appartments, ⋯
    &appartment_t::monthly_payment);
```
Pointers to member functions

\[
\begin{align*}
\text{accumulate(appartments, } & \text{compose(&tenant_t::monthly_payment, } \\
& \text{&appartment_t::tenant));}
\end{align*}
\]
Pointers to member objects

&something_t::m_full_name
Pointers to member objects

&something_t::m_full_name :
    something_t -> std::string
Pointers to member objects

&something_t::m_full_name : something_t -> std::string
(something_t, std::string) -> something_t
UNITE
Pairing up

Getters and setters

*Note: Not really OOP...*
Pairing up

Getters and updater

Note: Not really OOP...
Pairing up

view : object_t -> value_t

update : object_t, (value_t -> value_t) -> object_t
template <typename Object, typename Value, typename View, typename Update>
class typed_property {
    ...
};
Pairing up

template <typename Object, typename Value, 
     typename View, typename Update>

class typed_property {
public:
    using object_t = Object;
    using value_t = Value;

private:
    View m_view;
    Update m_update;
};
Pairing up

deftype(auto) view(const object_t &object) const
{
    return std::invoke(m_view, object);
}
Pairing up

decltype(auto) update(object_t &&object, auto updateFn) const
{
    return std::invoke(m_update, std::move(object), updateFn);
}
Pairing up

decltype(auto) set(object_t &&object, auto value) const {
    return std::invoke(m_update, std::move(object),
                     [_value = std::move(value)] (auto&&) {
                         return _value;
                     });
}
### Pairing up

| operator() (const object_t&) | calls view |
| operator() (object_t&&, update_function) | calls update |
| operator() (object_t&&, value_t) | calls set |
namespace detail
{
    template<typename T, typename M>
    struct member_splitter_result {
        using object_t = T;
        using member_ptr_t = M;
    };

    template<typename T, typename M>
    member_splitter_result<T, M> member_splitter_helper(M(T::*m));

    template<typename MPtr>
    using member_traits = decltype(member_splitter_helper(std::declval<MPtr>()));
} // namespace detail
Pairing up

template<typename MemberPtr>
auto member(MemberPtr member)
{
    using object_t = ...; using member_value_t = ...;

    auto update = [member](object_t &&object, auto &&updateFn) -> object_t&&
    {
        auto &field = std::invoke(member, object);
        field = std::invoke(updateFn, std::move(field));
        return std::move(object);
    };

    return typed_property<
        object_t, member_value_t,
        MemberPtr, decltype(update)>(member, std::move(update));
}
How can we compose properties?

- View
- Update
Back to composition
Back to composition
Back to composition
Back to composition

Composition
Unite
Abstraction
The End

Members

Unite

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End

Composition

Members

Unite

Abstraction

The End
auto compose_update =
    [ _left = left.update_function(),
      _right = right.update_function()](outer_object_t outerValue, auto innerUpdateFn) {
        auto outerUpdateFunction =
            [ _left, innerUpdateFn](inner_object_t innerValue) {
                return std::invoke(_left, std::move(innerValue),
                        innerUpdateFn);
            };

        return std::invoke(_right, std::move(outerValue),
                        std::move(outerUpdateFunction));
    };

Back to composition
Back to composition
Back to composition
auto monthly =
   fn::tenant >> fn::monthly_payment;

accumulate(
   appartments, 0.0, std::plus{}, monthly);
```cpp
class Account {
    std::string name;
    double balance;

public:
    Account(std::string name, double balance) : name(name), balance(balance) {}
    void deposit(double amount) { balance += amount; }
    void withdraw(double amount) { if (amount <= balance) balance -= amount; }
    double getBalance() const { return balance; }
};
```

```cpp
auto monthly = fn::tenant >> fn::monthly_payment;
accumulate(appartments, 0.0, std::plus{}, monthly);
new_appartment = monthly(std::move(old_appartment), increase_20_percent);
```
ABSTRACTION
But, what is composition?

- Functions
- Properties
...
But, what is composition?

\[
f: \text{object}_t \rightarrow \text{optional}<\text{value}_t>;
g: \text{value}_t \rightarrow \text{optional}<\text{inner}_t>;
\]
\[
\text{compose}(g, f): \text{object}_t \rightarrow \text{optional}<\text{inner}_t>;
\]
But, what is composition?

\[
f: \text{object}_t \rightarrow \text{vector}<\text{value}_t>;
g: \text{value}_t \rightarrow \text{vector}<\text{inner}_t>;
\]

\[
\text{compose}(g, f): \text{object}_t \rightarrow \text{vector}<\text{inner}_t>;
\]
But, what is composition?

```cpp
building_t::appartments : building_t -> vector<appartment_t>;
appartment_t::tenant : appartment_t -> tenant_t;
tenant_t::monthly_payment : tenant_t -> double

auto payments = fn::appartments
     >> fn::tenant
     >> fn::monthly_payment;
accumulate(payments(building), 0.0);
```
But, what is composition?

```cpp
building_t::appartments : building_t -> vector<appartment_t>;
apartment_t::tenants : appartment_t -> vector<tenant_t>;
tenant_t::monthly_payments : tenant_t -> vector<double>

auto payments = fn::appartments
                >> fn::tenants
                >> fn::monthly_payments;
accumulate(payments(building), 0.0);
```
But, what is composition?

```cpp
building_t::appartments : building_t -> vector<appartment_t>
appartment_t::tenants : appartment_t -> vector<tenant_t>
tenant_t::monthly_payments : tenant_t -> vector<double>

auto payments = fn::appartments
  >> fn::tenants
  >> fn::monthly_payments;
auto expensive_building =
payments(std::move(building), increase_20_percent);
```
Focus
Focus

fn::appartments

- building
- appartments
- tenant
- monthly_payment
Focus

fn::appartments >> fn::tenants

- building
- appartment
- tenant
- monthly_payment
Focus

```cpp
fn::appartments >> fn::tenants >> fn::monthly_payments
```

- building
- appartment
- tenant
- monthly_payment
accumulate(payments(building), ...)
auto expensive_building =
payments(std::move(building), ...)
Composition in the future

\[
f: \text{object}_t \rightarrow \text{future}<\text{value}_t>;
g: \text{value}_t \rightarrow \text{future}<\text{inner}_t>;
\]

\[
\text{compose}(g, f): \text{object}_t \rightarrow \text{future}<\text{inner}_t>;
\]
Composition in the future

```cpp
building_t::appartments : building_t -> future<appartment_t>;
appartment_t::tenants : appartment_t -> future<tenant_t>;
tenant_t::monthly_payments : tenant_t -> future<double>

auto payments = fn::appartments
    >> fn::tenants
    >> fn::monthly_payments;
not_really_accumulate(payments(building), 0.0);
```
Wrap

- Core-language support is great, but...
- Refactoring
- Easily expanded to filtering, optional values, ...
- Nice for building GUIs
Functional Programming in C++
cukic.co/to/fp-in-cpp

Personal
Web: https://cukic.co
Mail: ivan@cukic.co
Twitter: @ivan_cukic

Web: https://kdab.com
Mail: ivan.cukic@kdab.com