KDAB

Compile-time type transformation

Meeting C++ 2019, Berlin

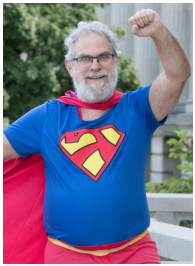
dr Ivan Čukić

KDAB
ivan.cukic@kdab.com, ivan@cukic.co
https://kdab.com, https://cukic.co

About me

- KDAB senior software engineer
  Software Experts in Qt, C++ and 3D / OpenGL
- Trainer / consultant
- KDE developer
- Author of the ”Functional Programming in C++” book
- University lecturer

Disclaimer



Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.

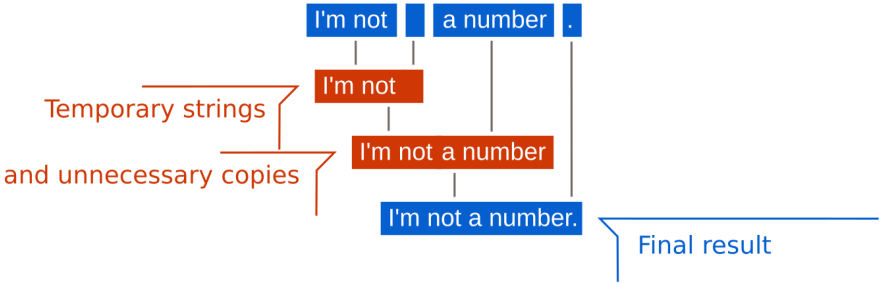
Philip Wadler

SLOW INTRODUCTION

String concatenation

```
std::string statement{”I'm not”};
std::string number{”a number”};
std::string space{” ”};
std::string period{”. ”};

std::string result = statement + space + number + period;
```

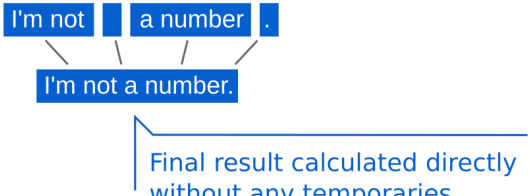
String concatenation



String concatenation

```
std::string result;
result.reserve(statement.size() + space.size()
               + number.size() + period.size());
result.append(statement);
result.append(space);
result.append(number);
result.append(period);
```

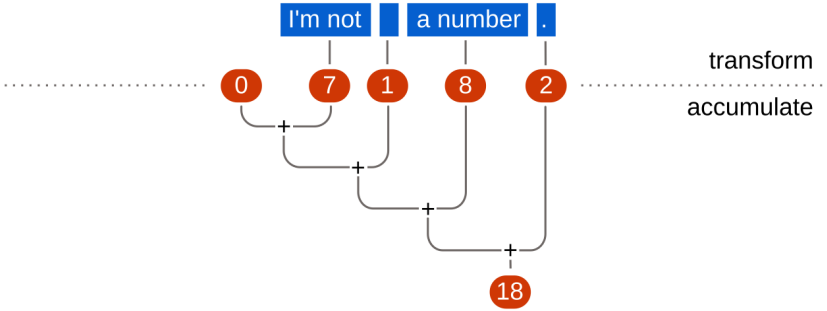
String concatenation



String concatenation

```
std::string concatenate(const std::vector<std::string>& strs)
{
    std::string result;

    result.reserve(accumulate(
        strs | transform(&std::string::size), 0));

    for (const auto& str: strs) { … }

    return result;
}
```

String concatenation



String concatenation

```
std::string concatenate(const std::vector<std::string>& strs)
{
    std::string result;

    result.reserve(accumulate(
        strs | transform(&std::string::size), 0));

    for (const auto& str: strs) {
        result.append(str);
    }


    return result;
}
```

String concatenation

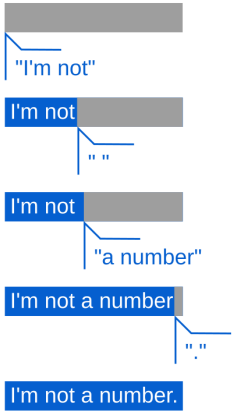
```
std::string concatenate(const std::vector<std::string>& strs)
{
    std::string result;

    result.resize(accumulate(
        strs | transform(&std::string::size), 0));


    ???



    return result;
}
```

String concatenation

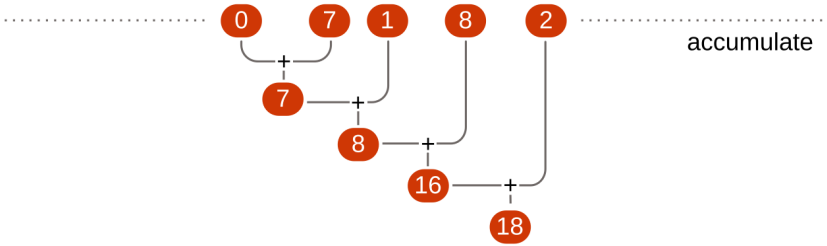
```
std::string concatenate(const std::vector<std::string>& strs)
{
    std::string result;

    result.resize(accumulate(
        strs | transform(&std::string::size), 0));

    accumulate(strs, result.begin(),
               [] (const auto& dest, const std::string& s) {
                   return copy(s, dest);
               });

    return result;
}
```

String concatenation

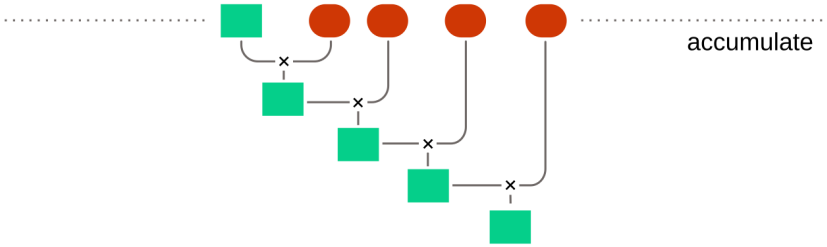
Each call to copy returns the iterator to the place after the last copied character -- the position that we should copy the next string to



Accumulate



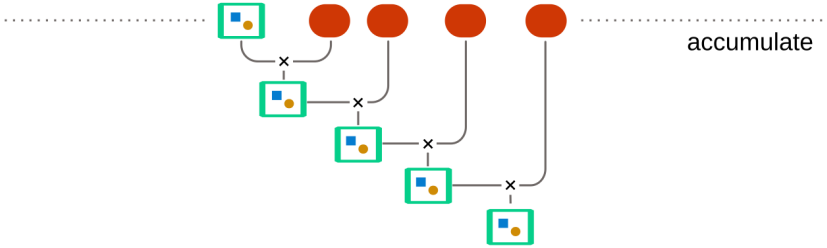
Accumulate



Accumulate

What if we want to save all the intermediate iterators into a vector?

Accumulate



Accumulate

Accumulator type:

```
struct acc_t {
    std::string::iterator dest;
    std::vector<std::string::iterator> points;
};
```

Accumulate

Initial accumulator value:

```
acc_t {
    result.begin(),
    {}
};
```

Accumulate

Accumulation function:

```
[] (acc_t&& acc, const std::string& s)
{
    acc.points.push_back(acc.dest);
    return acc_t {
        copy(s, acc.dest),
        std::move(acc.points)
    };
}
```

Accumulate

```
std::string concatenate(const std::vector<std::string>& strs)
{
    …

    struct acc_t {
        std::string::iterator dest;
        std::vector<std::string::iterator> points;
    };

    auto acc = accumulate(strs, acc_t { result.begin(), {} },
               [] (acc_t&& acc, const std::string& s)
               {
                   acc.points.push_back(acc.dest);
                   return acc_t {
                       copy(s, acc.dest),
                       std::move(acc.points)
                   };
               });

    acc.dest …
    acc.points …
}
```

Accumulate

```
std::pair<std::string, std::vector<std::string_view>>
concatenate(const std::vector<std::string>& strs)
{
    ...
}
```

Note: Beware of `string_view` and UB

COMPILE-TIME

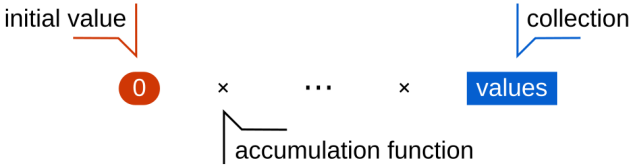
Compile-time

```
template <typename... Strings>
auto concatenate(Strings&&... strs)
{
    result.resize(accumulate(…));
}
```

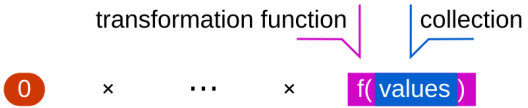
Compile-time

What is a compile-time accumulate?

Compile-time



Compile-time



Compile-time

```
template <typename... Strings>
auto concatenate(Strings&&... strs)
{
    const auto total_size = (0 + ... + strs.size());
    std::string result;
    result.resize(total_size);

    …

}
```

Compile-time

```
template <typename... Strings>
auto concatenate(Strings&&... strs)
{
    const auto total_size = (0 + ... + strs.size());
    std::string result;
    result.resize(total_size);

    (result.begin() << ... << strs);

}
```

Compile-time

```
template <typename Dest, typename String>
auto operator<< (Dest dest, const String& string)
{
    return copy(string, dest);
}
```

Compile-time

```
template <typename Init, typename Fun>
struct accumulator_t {
    Init value;
    const Fun& f;

    accumulator_t(Init value, const Fun& f)
        : value{value}, f{f}
    {}

    …



};
```

Compile-time

```
template <typename Init, typename Fun>
struct accumulator_t {
    …

    template <typename T>
    auto operator<< (T&& t) const
    {
        return accumulator_t(
            std::invoke(f, value, std::forward<T>(t)),
            f);
    }

    …
};
```

Compile-time

```
template <typename Init, typename Fun>
struct accumulator_t {

    …

    operator Init() const
    {
        return value;
    }
};
```

Compile-time

```
(accumulator_t(0, std::plus{}) << ... << strs.size());
```

Compile-time

```
auto add_size = [](int previous, const std::string &s)
{
    return previous + s.size();
};

(accumulator_t(0, add_size) << ... << strs);
```

Compile-time

```
template <typename Init, typename Fun, typename... Vals>
auto accumulate(Init&& init, const Fun& fun, Vals&&... vals)
{
    return (
        accumulator_t(std::forward<Init>(init), fun)
        << ...
        << std::forward<Vals>(vals)).value;
}
```

String concatenation

```
template <typename... Strings>
std::string concatenate(Strings&& strs)
{
    …

    result.resize(accumulate(0, add_size, strs...));

    accumulate(result.begin(),
               [] (const auto& dest, const std::string& s) {
                   return copy(s, dest);
               },
               std::forward<Strings>(strs)...);

    …
}
```

META INFORMATION

Meta information

```
print_animals(const std::vector<animal*> animals);

std::vector<dog*> dogs;

print_animals(dogs); // ERROR
```

Meta information

```
template < typename From
         , typename To



         >
concept covariant = …;
```

Meta information

```
template < typename From
         , typename To
         , typename FromTPtr = typename From::value_type
         , typename ToTPtr = typename To::value_type




         >
concept covariant = …;
```

Meta information

```
template < typename From
         , typename To
         , typename FromTPtr = typename From::value_type
         , typename ToTPtr = typename To::value_type
         , typename FromT = std::remove_cvref_t<
                                decltype(*std::declval<FromTPtr>())>
         , typename ToT = std::remove_cvref_t<
                                decltype(*std::declval<ToTPtr>())>
         >
concept covariant = std::is_base_of_v<ToT, FromT>;
```

Meta information

```
template <typename Collection>
         requires (covariant<Collection, std::vector<animal*>>)
void print_animals(const Collection& xs)
{
}

// or constexpr bool and enable_if
```

Meta information

```
void print_animals(
    covariant<std::vector<animal*>> auto const& xs)
{
}
```

Meta information

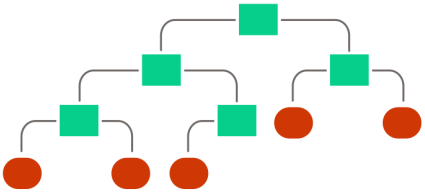
```
if constexpr (std::is_array_v<T>) {

} else {

}
```

Meta information

```
partition(ForwardIterator first, ForwardIterator last,
          Predicate pred, forward_iterator_tag)
partition(ForwardIterator first, ForwardIterator last,
          Predicate pred, bidirectional_iterator_tag)
```

GENERATION

Ropes



Ropes

- Template expressions are usually binary trees
- Rope leaves can be quite different in types and sizes
- The tree can be, but does not need to be balanced

Ropes

```
template <typename Left, typename Right>
struct node {
    Left left;
    Right right;









};
```

Ropes

```
template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_count =
        int_val<node_count_v<Left> +
                node_count_v<Right>>;

    …


};
```

Ropes

```
template <typename Left, typename Right>
struct node {
    …
    template <size_t I>
    const auto& node_at() const
    {
        if constexpr (I < node_count_v<Left>) {
            return left.node_at<I>();
        } else {
            return right.node_at<I - node_count_v<Left>>();
        }
    }
};
```

Ropes

```
template <typename Tree, size_t... Idx>
auto tree_to_tuple_helper(
        const Tree& tree,
        std::index_sequence<Idx...>)
    return std::make_tuple(tree.node_at<Idx>()...);
}
```

Ropes

```
template <typename Tree, size_t... Idx>
auto tree_to_tuple_helper(
        const Tree& tree,
        std::index_sequence<Idx...>)
    return std::make_tuple(tree.node_at<Idx>()...);
}

template <typename Tree>
auto tree_to_tuple(const Tree& tree)
{
    tree_to_tuple_helper(
        tree,
        std::make_index_sequence<node_count_v<Tree>>());
}
```

Ropes

```
template <typename Tree, size_t... Idx>
auto process(const Tree& tree, std::index_sequence<Idx...>)
    return (init << ... << tree.node_at<Idx>());
}
```

Ropes

```
template <typename... Ts>
struct leaf {
    std::tuple<Ts...> data;

    template <size_t I>
    const auto& node_at() const
    {
        return std::get<I>(data);
    }
}
```

TMP

So far, nothing new. Just *normal* TMP.

Ropes

```
struct binary_node_tag;

template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_type_tag = binary_node_tag;


};
```

Ropes

```
struct binary_node_tag;

template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_type_tag = binary_node_tag;
    using left_t = Left;
    using right_t = Right;
};
```

Ropes

```
if constexpr (
    is_detected_exact_v<
        binary_node_tag,
        has_node_type_tag,
        T>) {
    do something with T::left_t and T::right_t …
}
```

Ropes

- Unsafe
- Not scalable

Ropes

```
struct binary_node_tag;

template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_type_tag = binary_node_tag;
    // using left_t = Left;
    // using right_t = Right;
};
```

Ropes

```
template <typename Left, typename Right>
struct binary_node_tag { using left = Left; using right = Right; };

template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_type_tag = binary_node_tag<Left, Right>;


};
```

Ropes

Template instances as tags instead of types.

Ropes

Now, `is_detected_exact_v` is no longer an option

Ropes

```
template <typename Expected,
          template <typename...> typename Op,
          typename... Args>
using is_detected_exact =
          is_same<Expected, detected_t<Op, Args...>>;
```

Ropes

We need to check whether a given template has been used to instantiate a type:

`std::vector` is used for `std::vector<int>`
`std::basic_string` is used for `std::string`
`std::basic_string` is **not** used for `std::vector<char>`

Ropes

```
template <template <typename...> typename Template,
          typename Type>
struct is_instance_of: …
```

Ropes

```
template <template <typename...> typename Template,
          typename Type>
struct is_instance_of: std::false_type {};
```

Ropes

```
template <template <typename...> typename Template,
          typename Type>
struct is_instance_of: std::false_type {};

template <template <typename...> typename Template,
          typename... Args>
struct is_instance_of<Template, Template<Args...>>: std::true_type {};
```

Ropes

```
template <template <typename...> typename Template,
          typename Type>
struct is_instance_of: std::false_type {};

template <template <typename...> typename Template,
          typename... Args>
struct is_instance_of<Template, Template<Args...>>: std::true_type {};

template <template <typename...> typename Template,
          typename Type>
constexpr bool is_instance_of_v =
          is_instance_of<Template, Type>::value;
```

Ropes

```
template <template <typename...> typename Expected,
          template <typename...> typename Op,
          typename... Args>
using is_detected_instance_of =
          is_instance_of<Expected, detected_t<Op, Args...>>;
```

Ropes

```
is_detected_instance_of<binary_node_tag,
                        has_node_type_tag,
                        T>
```

Ropes

```
template <typename Left, typename Right>
struct binary_node_tag { using left = Left; using right = Right; };

template <typename Left, typename Right>
struct node {
    Left left;
    Right right;

    using node_type_tag = binary_node_tag<Left, Right>;


};
```

Extensibility

- Tags are fixed
- Extending a type is difficult

Extensibility

```
template <typename Left, typename Right, typename BaseMeta>
struct node {
    Left left;
    Right right;

    using meta_t =
        tuple_prepend_t<
            binary_node_tag<Left, Right>,
            BaseMeta
        >;
};
```

Extensibility

Does a type have a specific tag?

```
(false || ... || is_instance_of<specific_tag, Ts>);
```

Extensibility

Finding a specific tag:

```
if constexpr (is_instance_of<Tag, T>) {
    return tuple_append_t<Acc, T>{};
} else {
    return T{};
}
```

Extensibility

- We have a way to check for a tag (like `is_detected_v`)
- And we can get the tag (like `detected_t`)
- Difference – multiple instances of the same tag, tag meta-data

Extensibility

```
template <typename SourceNode>
struct enriched_node {
    using meta_t =
        tuple_prepend_t<
            enriched_node_tag<SourceNode>,
            some_common_tag<42>,
            typename SourceNode::meta
        >;
};
```

inheritance?

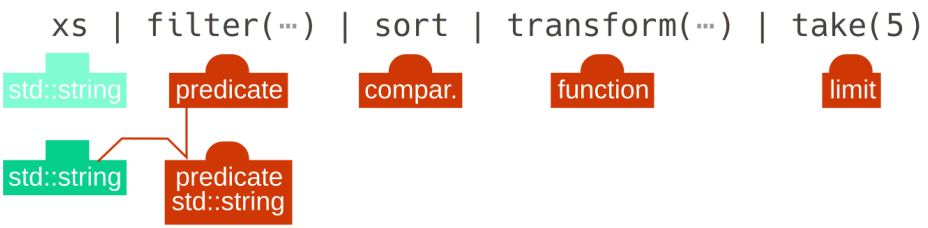
Pipes

```
xs | filter(…) | sort | transform(…) | take(5)
```

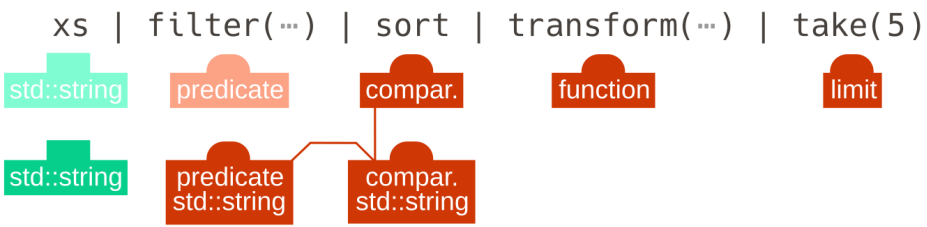
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit

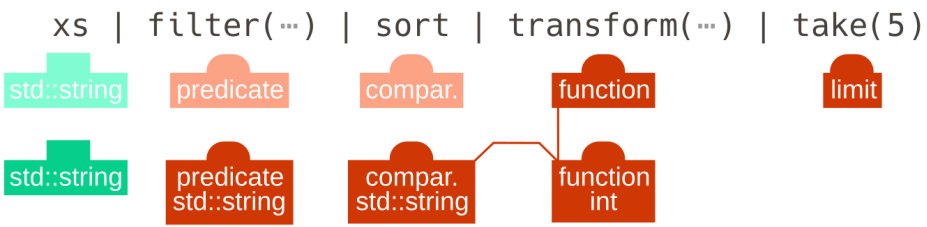
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string

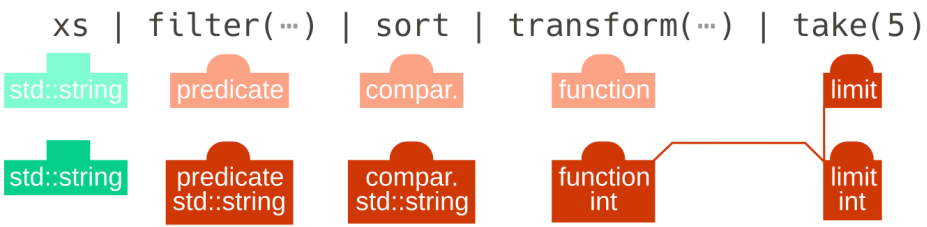
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string

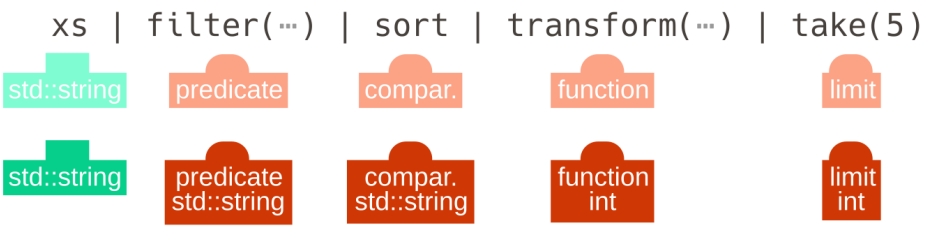
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int

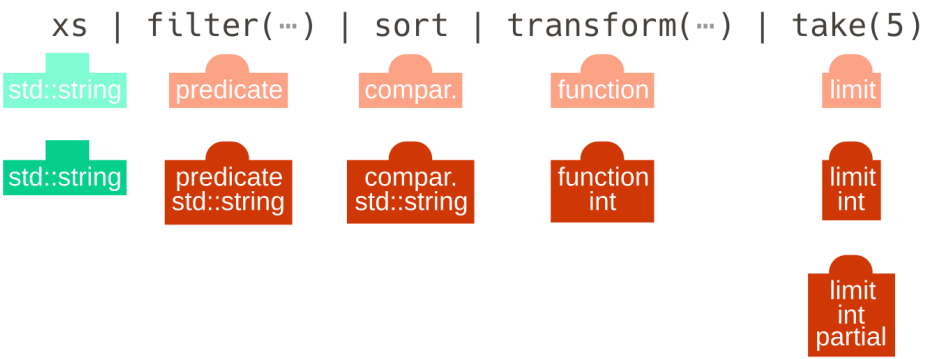
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int

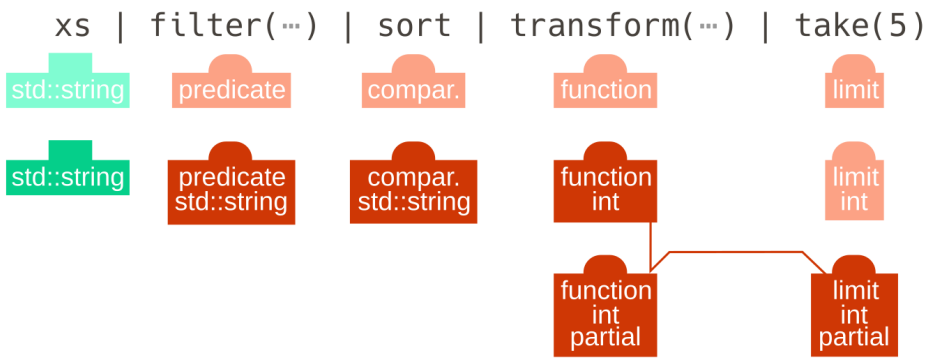
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int

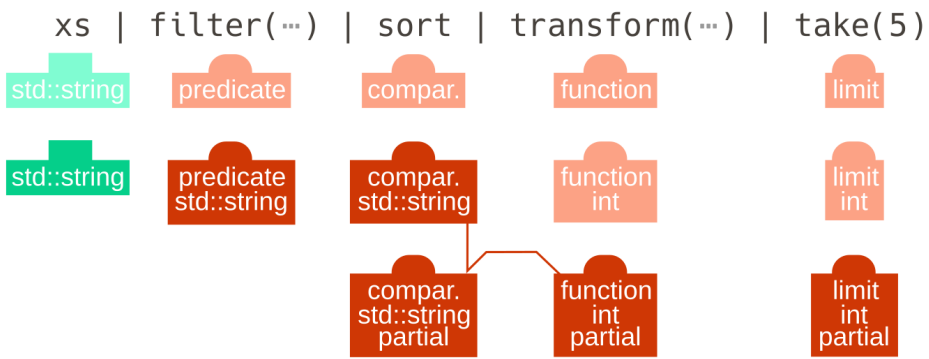
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int
limit
int
partial

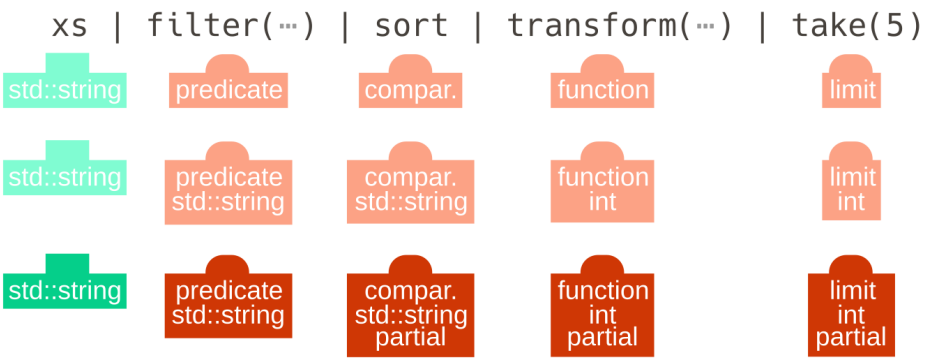
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int
function
int
partial
limit
int
partial

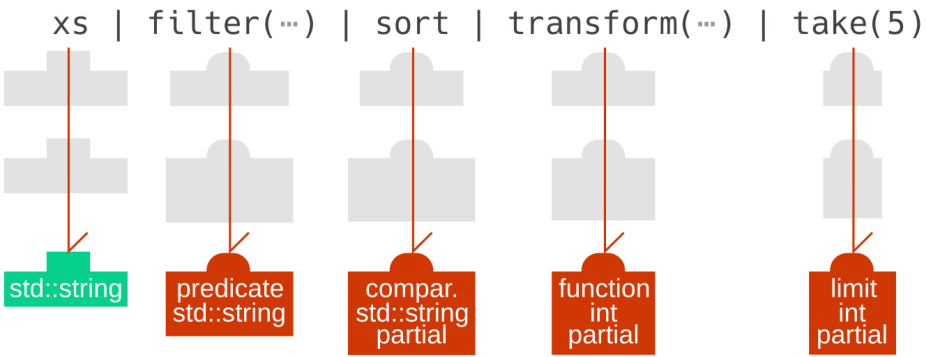
Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int
compar.
std::string
partial
function
int
partial
limit
int
partial

Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
compar.
function
limit
std::string
predicate
std::string
compar.
std::string
function
int
limit
int
std::string
predicate
std::string
compar.
std::string
partial
function
int
partial
limit
int
partial

Pipes


xs | filter(…) | sort | transform(…) | take(5)
std::string
predicate
std::string
compar.
std::string
partial
function
int
partial
limit
int
partial

Properties

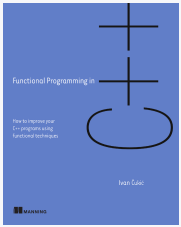
Dynamic property system...

at compile-time.

KDAB

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

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



Functional Programming in C++
cukic.co/to/fp-in-cpp