New C++ features for writing DSLs

ADC++ 2019, Regensburg

dr Ivan Čukić

ivan@cukic.co
http://cukic.co

About me

- Independent 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

INTRODUCTION

Introduction

```
select name from participants;

<expr> ::= <var> | <lit> | <expr> <op> <expr>

[a-zA-Z][a-zA-Z0-0_]*
```

DSLs and C++

Limited:

- `.something` syntax
- Operators
- Braces and parentheses

Introduction

<=>

Introduction

```
<_/\_~~*~~,,,,,\___/,,,>
```

BASICS

Basics first

```
user.name = ”Martha”;
user.surname = ”Jones”; // exception!
user.age = 42;
```

Copy-and-swap

```
type& operator=(const type& value)
{
    auto tmp = value;
    tmp.swap(*this);
    return *this;
}
```

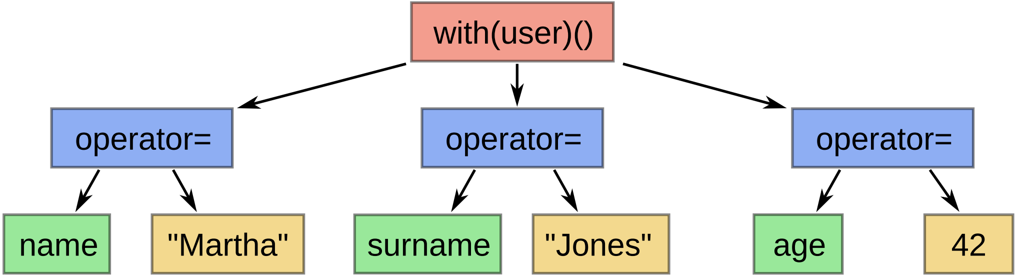
Basics first

```
auto tmp = user;
tmp.name = ”Martha”;
tmp.surname = ”Jones”;
tmp.age = 42;
tmp.swap(user);
```

Basics first

```
with(user) (
    name = ”Martha”,
    surname = ”Jones”,
    age = 42
);
```

Basics first



with(user)()

operator=

operator=

operator=

name

"Martha"

surname

"Jones"

age

42

Basics first

```
class transaction {
public:
    transaction(user_t& user)
        : m_user{user}
    {}

    void operator() (…)
    {
        …
    }

private:
    user_t& m_user;
};
```

Defines the object the transaction will operate on

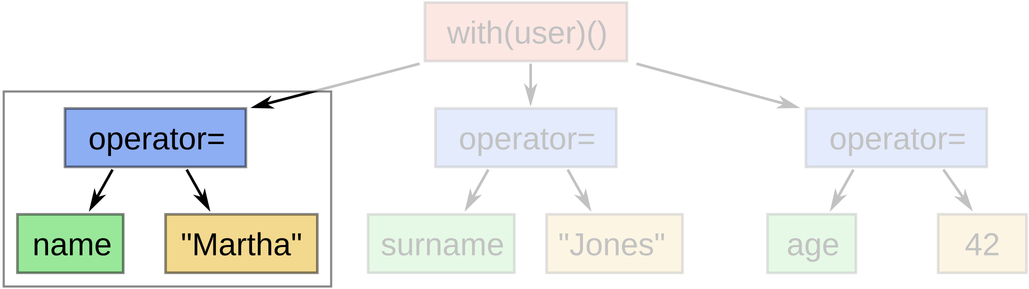
A reference to the object

Basics first

```
class transaction {
public:
    transaction(user_t& user)
        : m_user{user}
    {}

    void operator() (…)
    {
        …
    }

private:
    user_t& m_user;
};
```

Call operator takes
a list of actions
to perform

Basics first

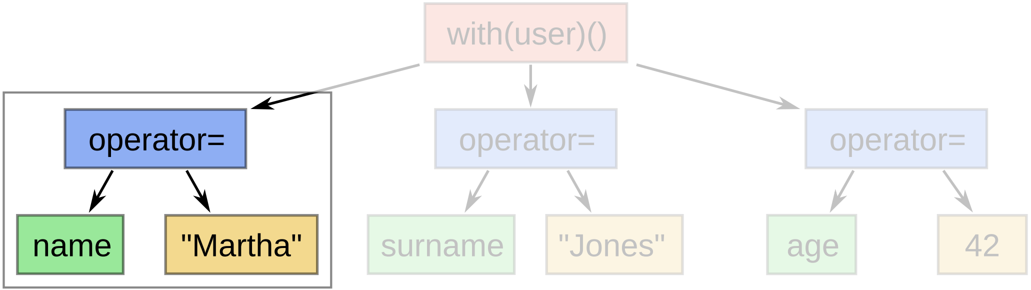


Basics first

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member{member}
        , value{std::move(value)}
    {
    }

    Member member;
    Value value;
};
```

Note that the update structure does not know which object it will be updating.

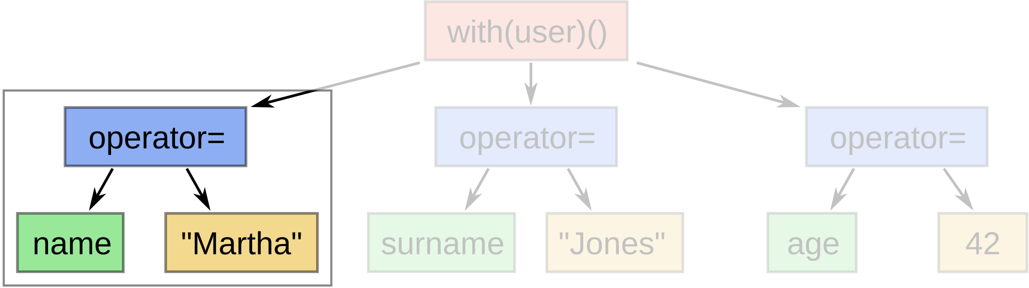
Basics first



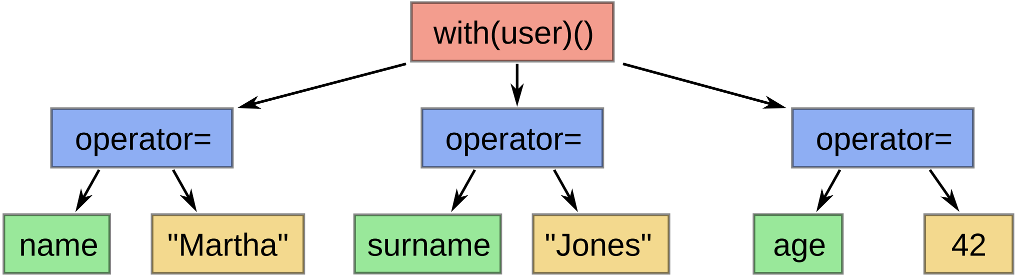
Basics first

```
template <typename Member>
struct field {
    field(Member member)
        : member{member}
    {
    }

    template <typename Value>
    update<Member, Value> operator=(Value&& value) const
    {
        return update{member, FWD(value)};
    }

    Member member;
};
```

Basics first



Basics first



Transaction

We’ll model a simple transaction concept:

- The update action is activated using the call operator:
- Each update action returns a `bool` indicating success or failure.

```
auto update_action{name = ”Martha”};

if (update_action(user)) {
    // success
}
```

Transaction

```
class transaction {
public:
    template <typename... Updates>
    bool operator() (Updates&&... updates)
    {
        auto temp = m_user;
```

> Invoke each update action on the temp object, and swap temp and *this only if they all succeeded

```
    }
};
```

Transaction

```
class transaction {
public:
    template <typename... Updates>
    bool operator() (Updates&&... updates)
    {
        auto temp = m_user;

        if ((... && FWD(updates)(temp))) {
            temp.swap(m_user);
            return true;
        }

        return false;
    }
};
```

Basics first

Ideal simple DSL:

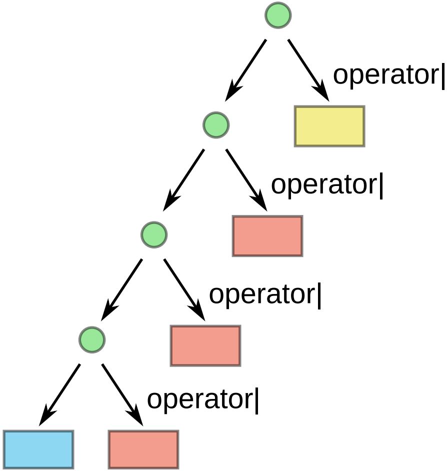
- Context-free
- AST that fits semantics
- Uses only simple constructs

CONTEXT

Ranges

```
users | transform(&user_t::name)
      | remove_if(&std::string::empty)
      | transform(string_to_upper);
```

Ranges



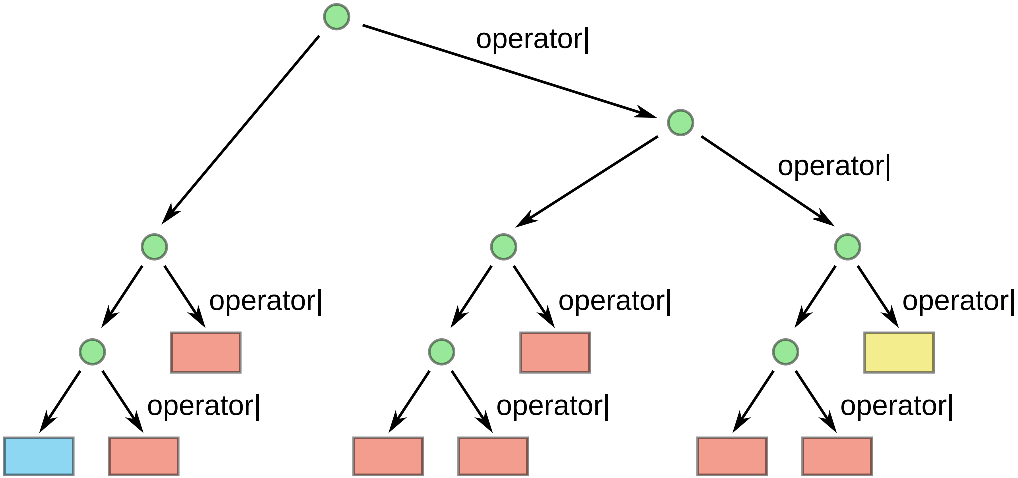
Ranges

```
template <typename... Nodes>
class expression {
    template <typename Transformation>
    auto operator| (Transformation&& trafo) &&
    {
        return expression(
            std::tuple_cat(
                std::move(m_nodes),
                std::make_tuple(FWD(trafo))));
    }

    std::tuple<Nodes...> m_nodes;
};
```

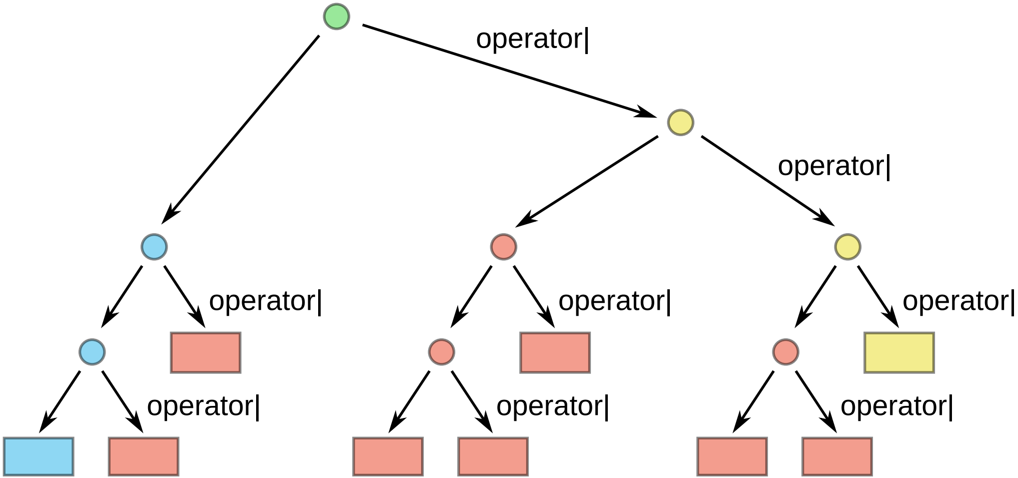
Ranges

```
auto user_names = users | transform(&user_t::name);
auto ignore_empty = transform(trim)
                  | remove_if(&std::string::empty);

user_names | ignore_empty | transform(string_to_upper);
```

Ranges


operator|
operator|
operator|
operator|
operator|
operator|
operator|
operator|

Ranges


operator|
operator|
operator|
operator|
operator|
operator|
operator|
operator|

Ranges

- Different meanings of `operator|`
- Wildly different types of operands (no inheritance tree)
- Arbitrary complex AST

Universal expression

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

```
<node> ::= <producer> | <consumer> | <trafo> | <expression>
<expression> ::= <node> <|> <node>
```

Meta information

Adding meta-information to classes:

```
struct producer_node_tag {};
struct consumer_node_tag {};
struct transformation_node_tag {};
```

```
class filter_node {
public:
    using node_type_tag =
        transformation_node_tag;
};
```

Meta information

```
template <typename Node>
using node_category =
    typename remove_cvref_t<Node>::node_type_tag;
```

Universal expression

```
template <typename Tag, typename Left, typename Right>
struct expression {
    using node_type_tag = Tag;

    Left left;
    Right right;
};
```

Meta information

```
template < typename Node
         , typename Category =
               std::detected_t<node_category, Node>
constexpr bool is_node()
{
    if constexpr (!is_detected_v<node_category, Node>) {
        return false;

    } else if constexpr (std::is_same_v<void, Category>) {
        return false;

    } else {
        return true;
    }
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    ...







}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    if constexpr (!is_producer<Left> && !is_consumer<Right>) {
        return expression<transformation_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }

    ...
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    … else
    if constexpr (is_producer<Left> && !is_consumer<Right>) {
        return expression<producer_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }
    …
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{
    … else
    if constexpr (!is_producer<Left> && is_consumer<Right>) {
        return expression<consumer_node_tag, Left, Right>{
            FWD(left), FWD(right)
        };
    }
    …
}
```

Restricting the pipe

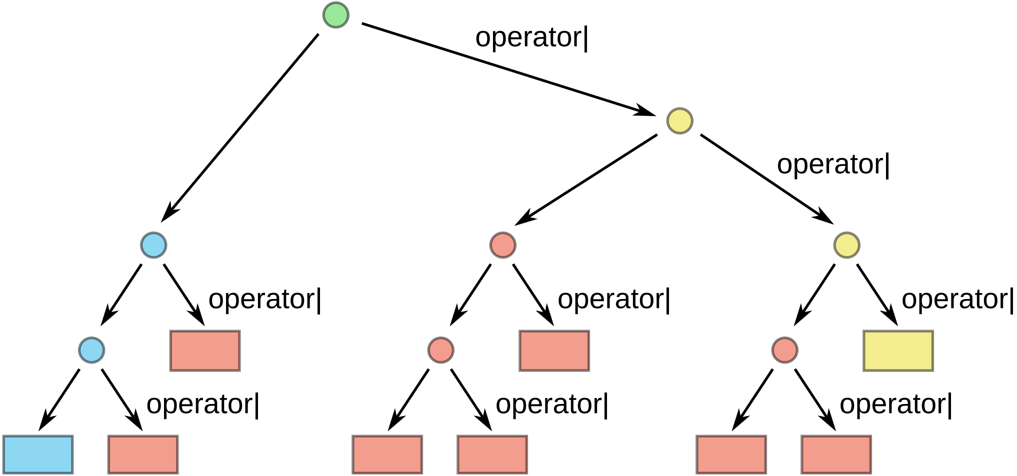
```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{

    … else
    if constexpr (is_producer<Left> && is_consumer<Right>) {
        return expression<void, Left, Right>{
            FWD(left), FWD(right)
        };
    }
}
```

Restricting the pipe

```
template < typename Left
         , typename Right
         , REQUIRE(is_node<Left>() && is_node<Right>())
         >
auto operator| (Left&& left, Right&& right)
{

    … else
    if constexpr (is_producer<Left> && is_consumer<Right>) {
        return evaluate(expression<void, Left, Right>{
            FWD(left), FWD(right)
        });
    }
}
```

EVALUATION

Evaluation


operator|
operator|
operator|
operator|
operator|
operator|
operator|
operator|

Evaluation

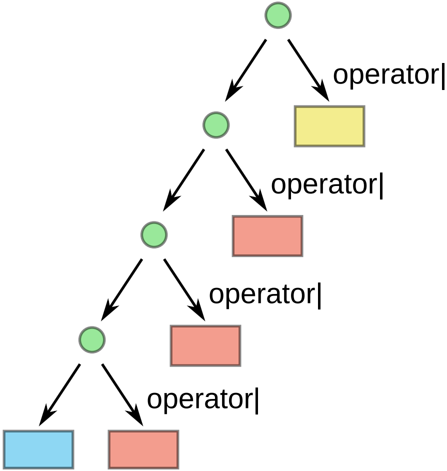
Range-like – pull semantics:

1. Consumer asks for a value
2. The request goes to the preceding transformation

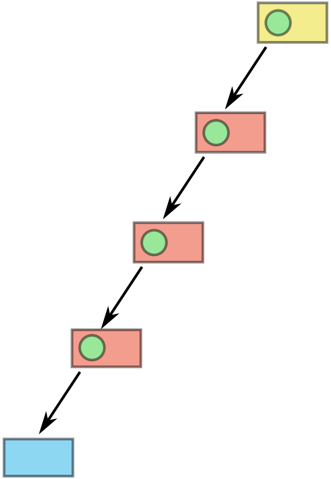
   …

nth The request gets to the producer

Ranges



Ranges



AST transformation

1. Collect nodes from the left sub-tree
2. Collect nodes from the right sub-tree
3. Merge the results

AST transformation

```
template <typename Expr>
auto collect_nodes(Expr&& expr)
{
    auto collect_sub_nodes = [] (auto&& sub) {
        if constexpr (is_expression<decltype(sub)>) {
            return collect_nodes(std::move(sub));
        } else {
            return std::make_tuple(std::move(sub));
        }
    };

    return std::tuple_cat(
        collect_sub_nodes(std::move(expr.left)),
        collect_sub_nodes(std::move(expr.right)));
}
```

Evaluation

We can use fold expressions yet again.

Fold expressions work on operators, so we need to create an operator instead of an eval function.

```cpp
template <typename Evaluated, Node new_node>
auto operator% (Evaluated&& evald, Node&& new_node)
{
    return FWD(new_node).with_producer(FWD(evald));
}
```

Evaluation

```
template <typename... Nodes>
auto evaluate_nodes(Nodes&&... nodes)
{
    return (... % nodes);
}
```

Evaluation

```
template <typename Tuple>
auto evaluate_nodes(Tuple&& nodes)
{
    return (... % std::get<?>(nodes));
}
```

Evaluation

```
template <typename Tuple, size_t... Idx>
auto evaluate_nodes_impl(Tuple&& nodes,
                         std::index_sequence<Idx...>)
{
    return (... % std::get<Idx>(nodes));
}
```

Evaluation

```
template <typename Tuple>
auto evaluate_nodes(Tuple&& nodes)
{
    return evaluate_nodes_impl(FWD(nodes),
        std::make_index_sequence<
            std::tuple_size_v<Tuple>>());
}
```

BEST PRACTICES

Asserts

```
#define assert_value_type(T)                        \
    static_assert(                                  \
        std::is_same_v<T, std::remove_cvref_t<T>>,  \
        ”This is not a value type”)
```

Printf debugging

```
template <typename... Types>
class print_types;
```

```
print_types<std::vector<bool>::reference>{};

error: incomplete type 'class print_types<std::_Bit_reference>'
```

Printf debugging

```
template <typename... Types>
[[deprecated]] class print_types;
```

Printf debugging

For complex expression template types, create a sanitization script:

- `basic_string...` → `string`
- `transformation_node_tag` → `TRAFO`

Change all < and > into `(` and `)` and pass the output through `clang-format`.

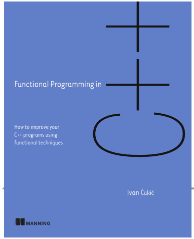
Printf debugging

```
expression(
    expression(
        void,
        expression(PRODUCER,
                   expression(PRODUCER, ping_process,
                              transform(”(λ tests_multiprocess.cpp:91:26)”)),
                   transform(”(λ tests_multiprocess.cpp:82:38)”))),
    expression(
        TRAFO,
        expression(TRAFO,
                   expression(TRAFO,
                              expression(TRAFO, identity_fn,
                                         transform(”(λ tests_multiprocess.cpp:99:

        ...
```

Answers? Questions! Questions? Answers!

Kudos (in chronological order):

Friends at **KDE**
Saša Malkov and **Zoltan Porkolab**
Сергей Платонов


Functional Programming in
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING

MANNING PUBLICATIONS

Functional Programming in C++

cukic.co/to/fp-in-cpp