Async. programming with ranges

Corehard Autumn 2018

dr Ivan Čukić

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

core-dev @ KDE
www.kde.org

About me

- KDE development
- Talks and teaching
- Author of ”Functional Programming in C++” (Manning)
- Functional programming enthusiast, but not a purist

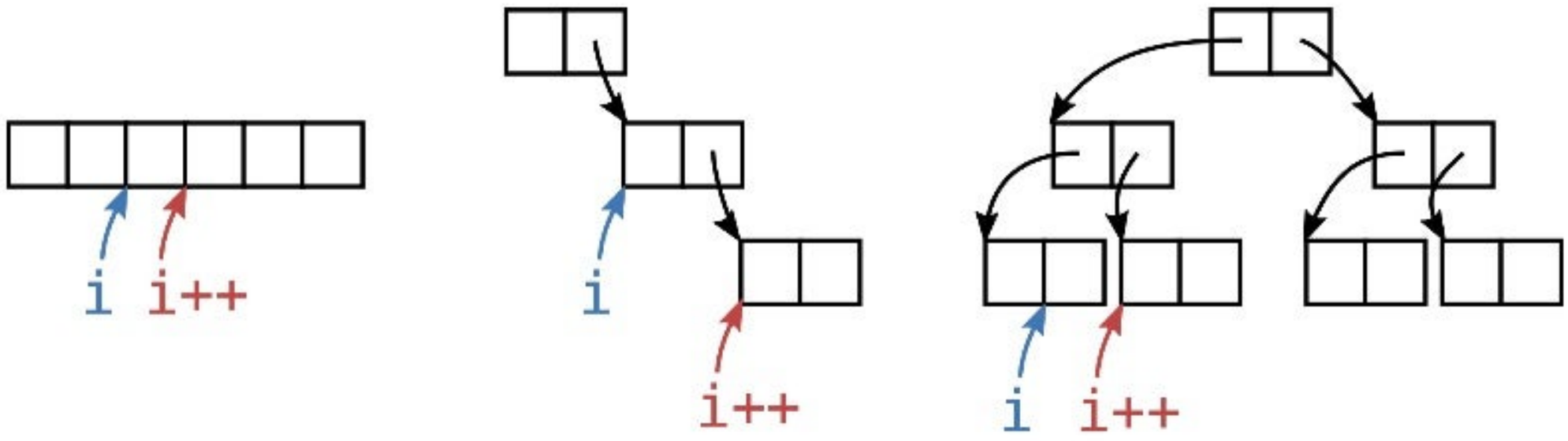
Disclaimer

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

ITERATORS

Iterators

At the core of generic programming in STL.



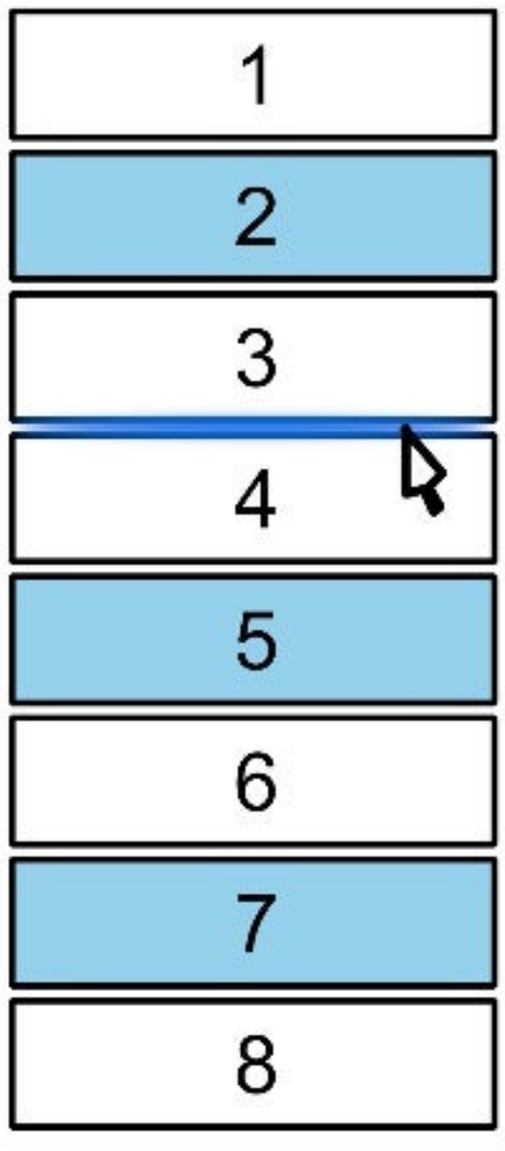
Iterators

```
std::vector<std::string> identifiers;

std::copy_if(
std::cbegin(tokens),
std::cend(tokens),
std::back_inserter(identifiers),
is_valid_identifier);
```

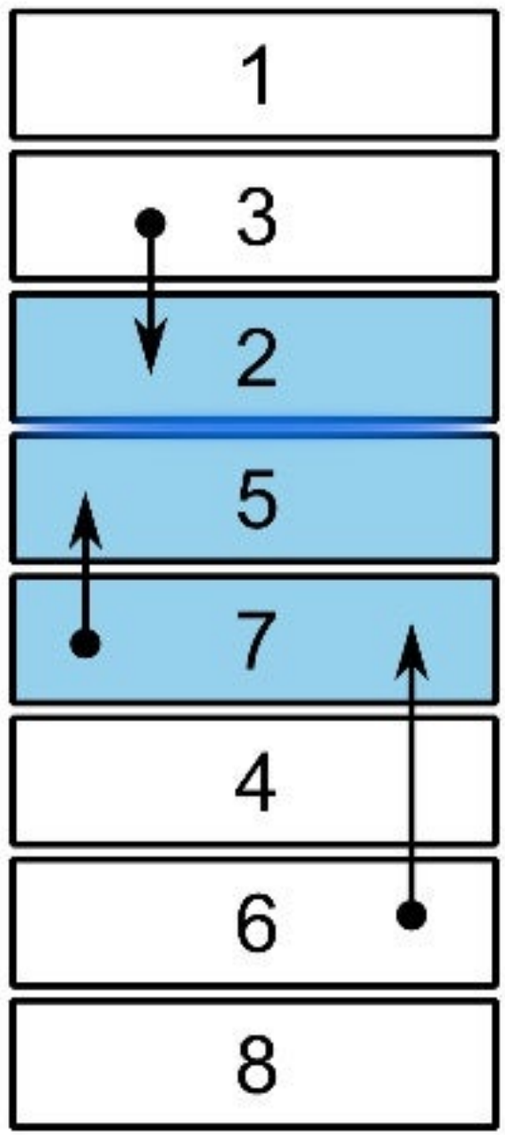
Iterators

```
std::vector<std::string> identifiers;

std::copy_if(
std::istream_iterator<std::string>(std::cin),
std::istream_iterator<std::string>(),
std::back_inserter(identifiers),
is_valid_identifier);
```

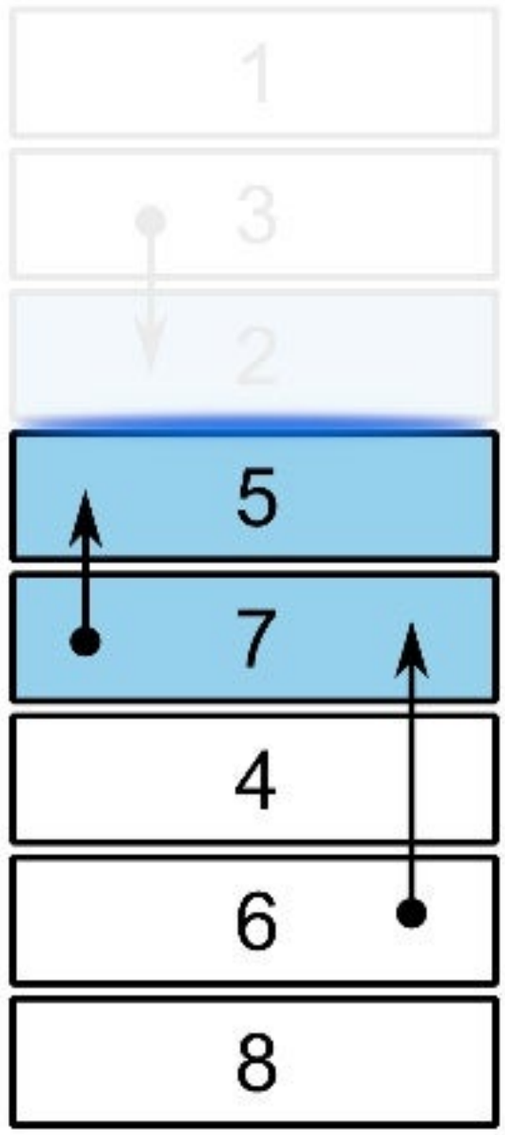
Example



Example



Example



Example

Example

```
template <typename It, typename Pred>
std::pair<It, It> move_selected(
    It begin, It end, It destination,
    Pred is_selected)
{
    return {
        std::stable_partition(begin, destination,
                              std::not_fn(is_selected)),
        std::stable_partition(destination, end,
                              is_selected)
    };
}
```

RANGES

Problems with iterators

Example: Sum of squares

This should be separable into two subtasks:

- squaring the values
- summation

Problems with iterators

```
std::vector<int> xs { ⋯ };

std::vector<int> squared(xs.size());
std::transform(std::cbegin(xs), std::cend(xs),
               std::begin(squared),
               [] (int x) { return x * x; });

return std::accumulate(std::cbegin(squared),
                       std::cend(squared),
                       0);
```

Problems with iterators

```
std::vector<int> xs { … };

return std::transform_reduce(par,
            std::cbegin(xs), std::cend(xs),
            std::cbegin(xs),
            0);
```

Proxy iterators

```
template <typename Trafo, typename It>
class transform_proxy_iterator {
public:

    auto operator*() const
    {
        return transformation(
                        *real_iterator);
    }

private:
    Trafo transformation;
    It real_iterator;
};
```

Ranges

```
[ iterator, sentinel )
```

Iterator:

- `*i` – access the value
- `++i` – move to the next element

Sentinel:

- `i == s` – has iterator reached the end?

Ranges

```
std::vector<int> xs { … };

accumulate(
    transform(
        xs,
        [] (int x) { … }),
0);
```

Ranges

```
std::vector<int> xs { … };

accumulate(
    transform(
        filter(
            xs,
            [] (int x) { … }),
        [] (int x) { … }),
0);
```

Ranges

```
std::vector<int> xs { … };

accumulate(
    xs | filter([] (int x) { … })
       | transform([] (int x) { … }),
    0);
```

Word frequency

1986: Donald Knuth was asked to implement a program for the ”Programming pearls” column in the Communications of ACM journal.

The task: Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

Word frequency

1986: Donald Knuth was asked to implement a program for the ”Programming pearls” column in the Communications of ACM journal.

The task: Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

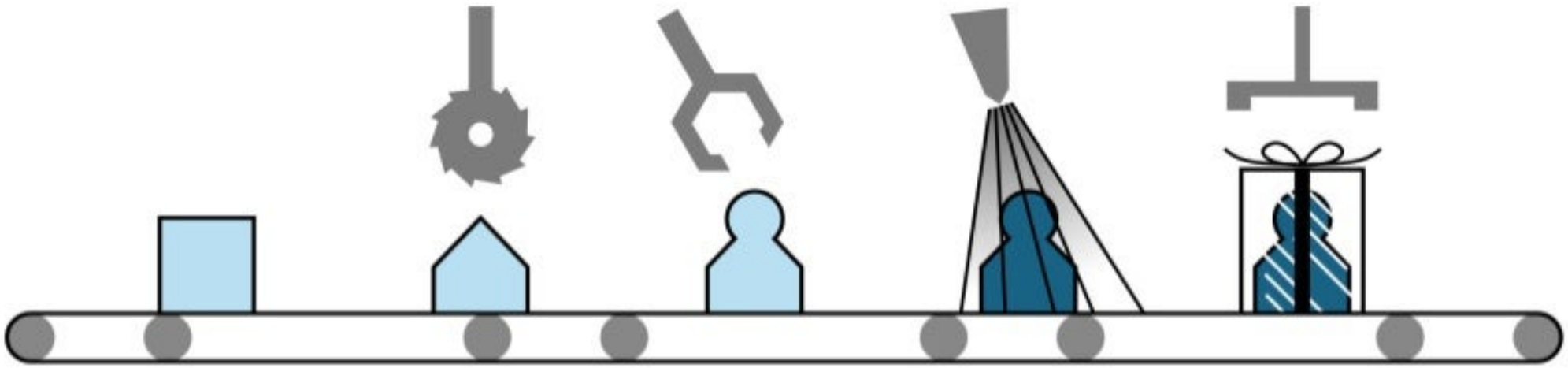
His solution written in Pascal was **10** pages long.

Word frequency

Response by Doug McIlroy was a 6-line shell script that did the same:

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

Functional thinking – data transformation



With ranges

```
const auto words =
    // Range of words
    istream_range<std::string>(std::cin)

    // Cleaning the input
  | view::transform(string_to_lower)
  | view::transform(string_only_alnum)
  | view::remove_if(&std::string::empty)

// Sort the result
| to_vector | action::sort;
```

With ranges

```
const auto results =
    words

    // Group repeated words
    | view::group_by(std::equal_to<>())

// Count how many repetitions we have
| view::transform([] (const auto &group) {
        return std::make_pair(
            distance(begin(group), end(group),
            *begin(group));
        })

// Sort by frequency
| to_vector | action::sort;
```

With ranges

```
for (auto value:
     results | view::reverse // descending
             | view::take(n) // we need first n results
     ) {
    std::cout << value.first << " "
              << value.second << std::endl;
}
```

PUSH

Ranges

```
[ iterator, sentinel )
```

Iterator:

- `*i` – access the value
- `++i` – move to the next element

Sentinel:

- `i == s` – has iterator reached the end

Ranges

```
[ iterator, sentinel )
```

Iterator:

- `*i` – access the value BLOCKING!
- `++i` – move to the next element

Sentinel:

- `i == s` – has iterator reached the end

Push iterators

Push iterators

Each *push iterator* can:

- Accept values
- Emit values

No need for the accepted and emitted values to be 1-to-1.

Push iterators

- Sources – push iterators that only emit values
- Sinks – push iterators that only accept values
- Modifiers – push iterators that both accept and emit values

Continuation

```
template <typename Cont>
class continuator_base {
public:
    void init() { … }

    template <typename T>
    void emit(T&& value) const
    {
        std::invoke(m_continuation, FWD(value));
    }

    void notify_ended() const { … }

protected:
    Cont m_continuation;
};
```

Invoke

```
std::invoke(function, arg1, arg2, ...)
```

For most cases (functions, function objects, lambdas) equivalent to:

```
function(arg1, arg2, ...)
```

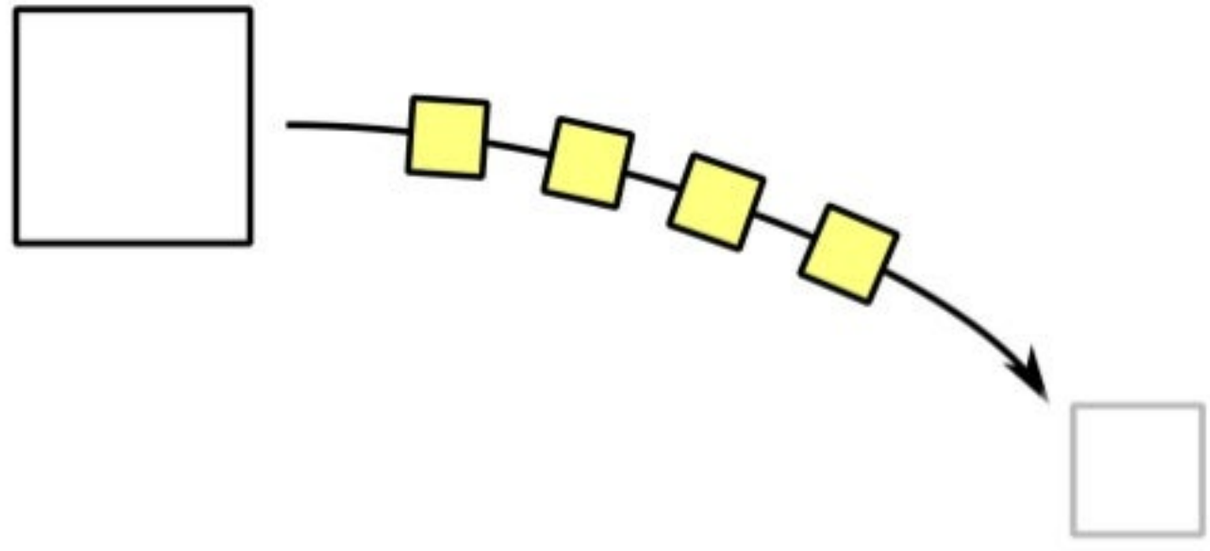
But it can also invoke class member functions:

```
arg1.function(arg2, ...)
```

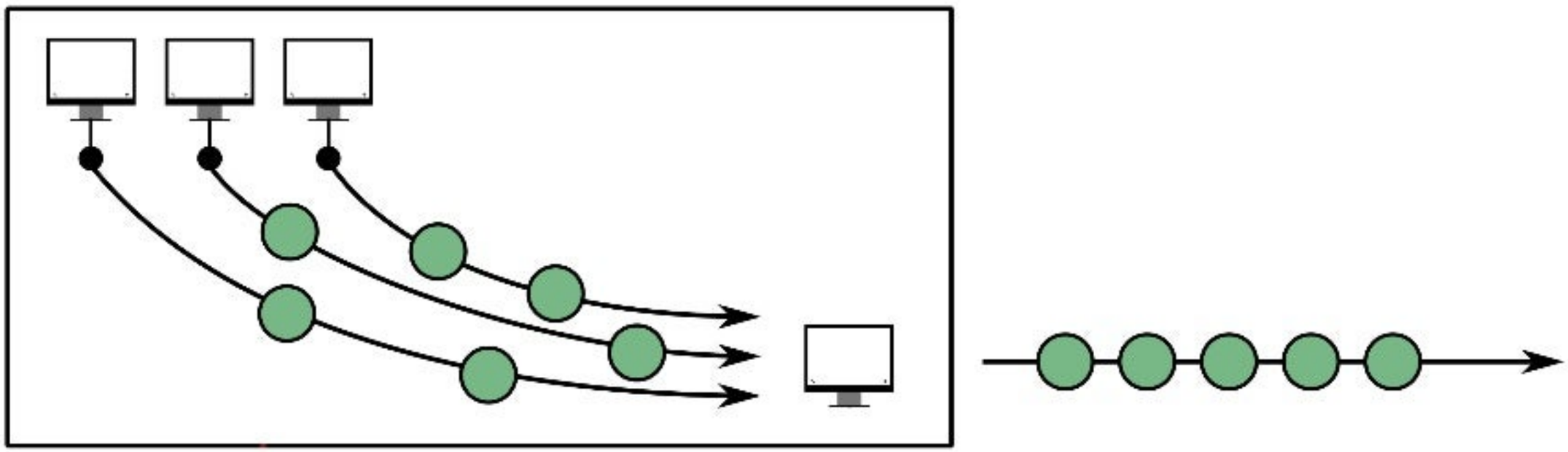
Source

```
template <typename Cont>
class values_node: public continuator_base<Cont>, non_copyable {
    …

    void init()
    {
        base::init();

        for (auto&& value: m_values) {
            base::emit(value);
        }

        m_values.clear();

        base::notify_ended();
    }

    …
};
```

Source

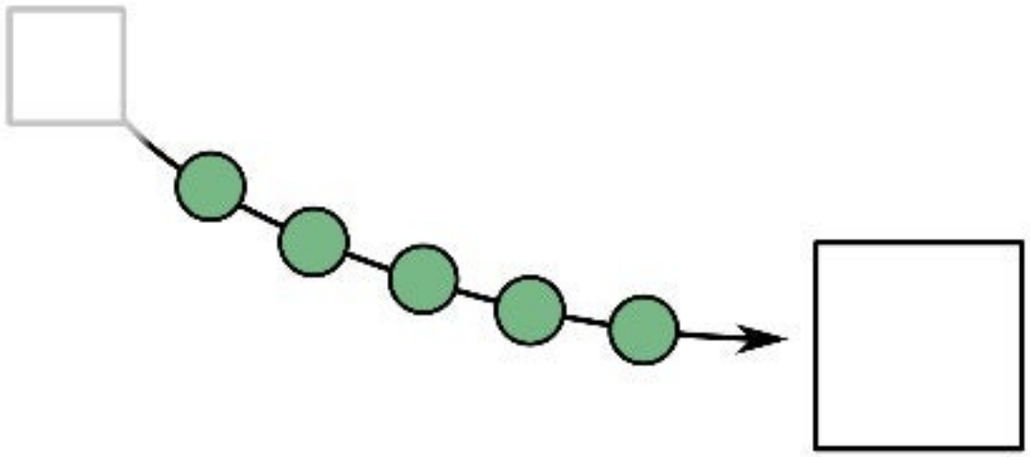


Creating a source

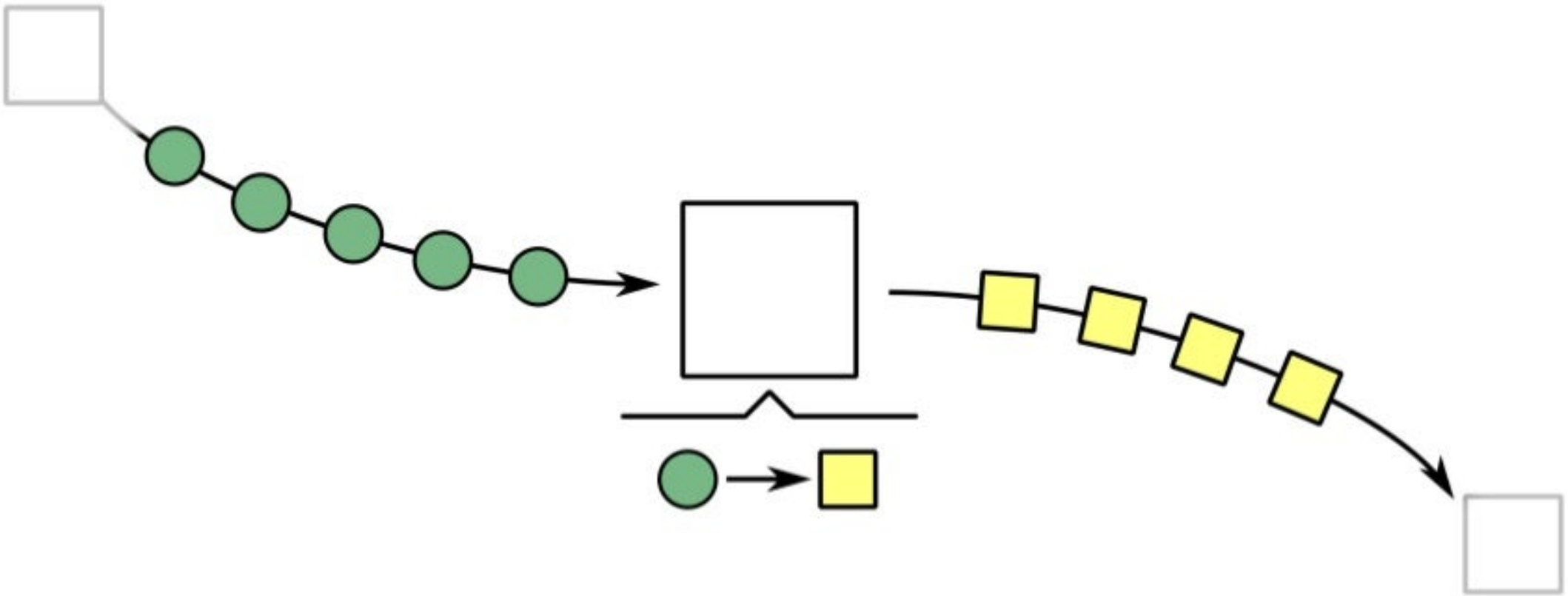


As far as the rest of the system is concerned, this node **creates** the messages.

Creating a sink



Creating a transformation



Creating a transformation

```
template <typename Cont>
class transform_node: public continuator_base<Cont>, non_copyable {
public:
    …
    template <typename T>
    void operator( ) (T&& value) const
    {
        base::emit(std::invoke(m_transformation, FWD(value)));
    }

private:
    Traf m_transformation;
};
```

Filtering

```
template <typename Cont>
class filter_node: public continuator_base<Cont>, non_copyable {
public:
    …

    template <typename T>
    void operator() (T&& value) const
    {
        if (std::invoke(m_predicate, value) {
            base::emit(FWD(value)));
        }
    }

private:
    Predicate m_predicate;
};
```

PIPES

Connecting the components

```
auto pipeline = source | transform(…)
                       | filter(…)
                       | join
                       | transform(…)
                       …
```

AST

When the pipeline AST is completed and evaluated, then we get proper pipeline nodes.

Associativity

```
auto my_transformation = filter(…)
                       | join;
```

```
auto pipeline = source | transform(…)
                       | my_transformation
                       | transform(…)
                       …
```

Different pipes

- Composing two transformations
  `transform(…) | filter(…)`
- Connecting a source to the continuation
  `source | transform(…)`
- Ending the pipeline
  `filter(…) | sink`
- Closing the pipeline
  `source | sink`

Different pipes

```
struct source_node_tag {};
struct sink_node_tag {};
struct transformation_node_tag {};
```

```
template <typename T>
class values {
public:
    using node_category = source_node_tag;

};
```

Different pipes

```
struct source_node_tag {};
struct sink_node_tag {};
struct transformation_node_tag {};
```

```
template <typename T>
class filter {
public:
    using node_category = transformation_node_tag;

};
```

Concepts

```
template <typename Left, typename Right>
auto operator| (Left&& left, Right&& right)
{
    return …;
}
```

Problem: Defines `operator|` on all types

Concepts

```
template < typename Node
         , typename Category = detected_t<node_category, Node>
         >
f_concept 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;
    }
}
```

Note: Proper concepts will come in C++20. This uses the detection idiom to simulate concepts.

Concepts

```
template < typename Left
         , typename Right
         , f_require(
                is_node<Left> && is_node<Right>
             )
         >
auto operator| (Left&& left, Right&& right)
{
    return …;
}
```

Concepts

```
if constexpr (!is_source<Left> && !is_sink<Right>) {
    ... create a composite transformation
} else if constexpr (is_source<Left> && !is_sink<Right>) {
    ...
} else if constexpr (!is_source<Left> && is_sink<Right>) {
    ...
} else {
    ...
}
```

Concepts

```
if constexpr (!is_source<Left> && !is_sink<Right>) {
    …
} else if constexpr (is_source<Left> && !is_sink<Right>) {
    …
} else if constexpr (!is_source<Left> && is_sink<Right>) {
    …
} else {
    …
}
```

transformed source is also a source

Concepts

```
if constexpr (!is_source<Left> && !is_sink<Right>) {
    …
} else if constexpr (is_source<Left> && !is_sink<Right>) {
    …
} else if constexpr (!is_source<Left> && is_sink<Right>) {
    …
} else {
    …
}
```

transformation with a sink is a sink

Concepts

```
if constexpr (!is_source<Left> && !is_sink<Right>) {
    ...
} else if constexpr (is_source<Left> && !is_sink<Right>) {
    ...
} else if constexpr (!is_source<Left> && is_sink<Right>) {
    ...
} else {
    ... the pipeline is complete, evaluate it
}
```

Pipes

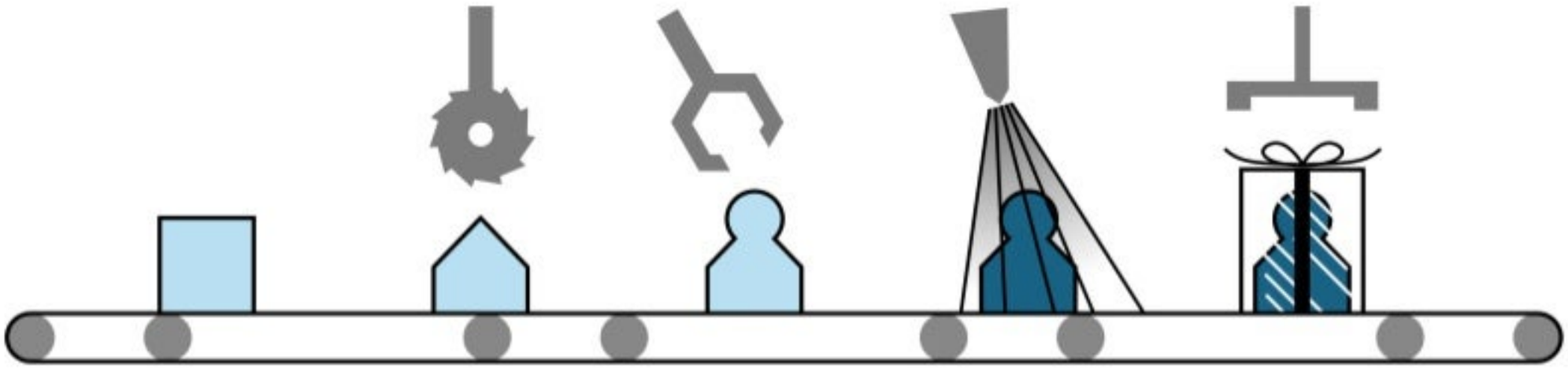
```
auto new_source = source | transform(…) | filter(…);

auto new_trafo = transform(…) | filter(…);

auto new_sink = filter(…) | sink;

auto pipeline = new_source | new_trafo | new_sink;
```

Pipes

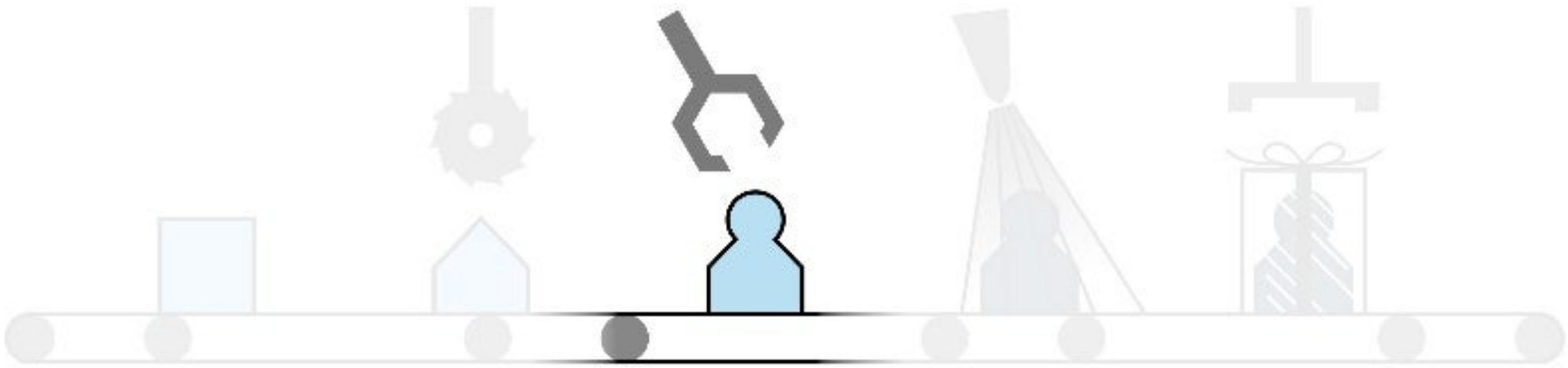
- Easy implementation of reactive systems
- Transparent handling of synchronous vs asynchronous transformations
- High-enough abstraction for more powerful idioms

GOING POSTAL

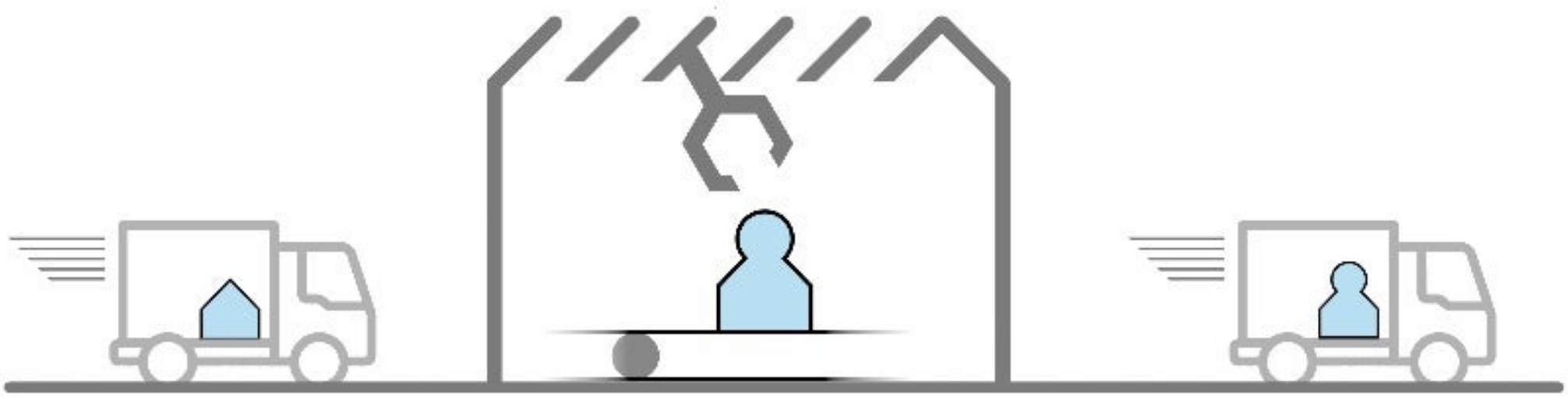
Functional thinking – data transformation



Functional thinking – data transformation



Functional thinking – data transformation



Distributed pipelines

```
auto pipeline =
    system_cmd(”ping”s, ”localhost”s)

    | transform(string_to_upper)

    // Parse the ping output
    | transform([] (std::string&& value) {
          const auto pos = value.find_last_of('=');
          return std::make_pair(std::move(value), pos);
      })

// Extract the ping time from the output
| transform([] (std::pair<std::string, size_t>&& pair) {
      auto [ value, pos ] = pair;
      return pos == std::string::npos
                  ? std::move(value)
                  : std::string(value.cbegin() + pos + 1, value.cend());
  })

// Remove slow pings
| filter([] (const std::string& value) {
      return value < ”0.145”s;
  })

// Print out the ping info
| sink{cout};
```

Distributed pipelines

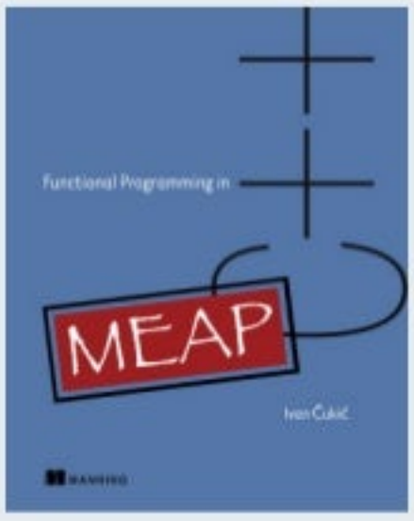
```
auto pipeline =
    system_cmd(”ping”s, ”localhost”s)

    | transform(string_to_upper)

    | voy_bridge(frontend_to_backend_1)

    | transform([] (std::string&& value) {
          const auto pos = value.find_last_of('=');
          return std::make_pair(std::move(value), pos);
      })

| transform([] (std::pair<std::string, size_t>&& pair) {
      auto [ value, pos ] = pair;
      return pos == std::string::npos
                  ? std::move(value)
                  : std::string(value.cbegin() + pos + 1, value.cend());
  })

| voy_bridge(backend_1_to_backend_2)

| filter([] (const std::string& value) {
      return value < ”0.145”s;
  })

| voy_bridge(backend_2_to_frontend)

| sink{cout};
```

Live demo

Answers? Questions! Questions? Answers!

Kudos (in chronological order):

Friends at **KDE**
Saša Malkov and **Zoltán Porkoláb**
спасибо **Сергею и двум Антонам**



MEAP – Manning Early Access Program
Functional Programming in C++
cukic.co/to/fp-in-cpp