The beast is becoming functional

CoreHard, Minsk 2017

Ivan Čukić

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

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

Connections

Functional programming

- Higher-order functions
- Algebraic data types
- Purity
- Laziness

etc.

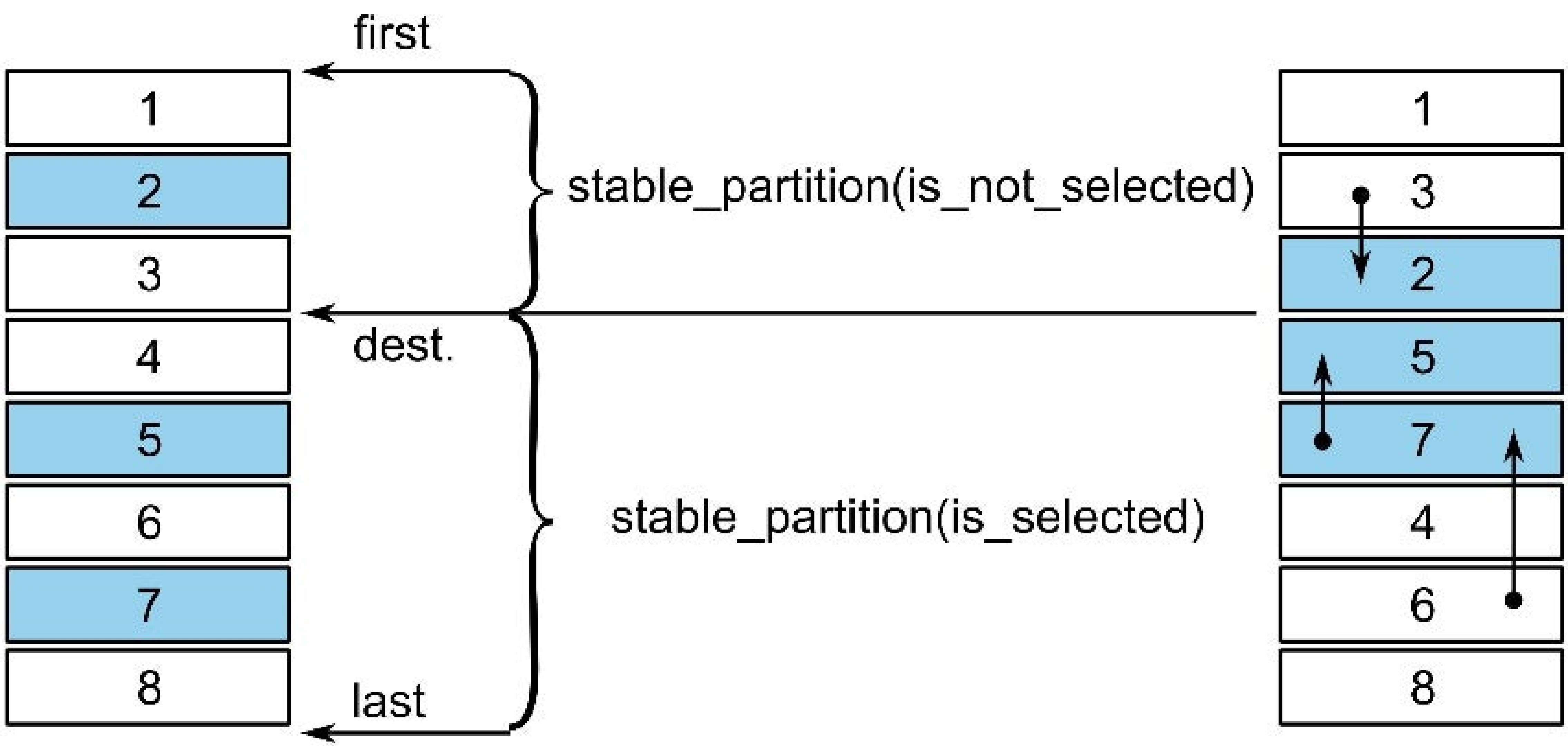
THE MIRACLE OF BIRTH

The Machine that Goes "Ping"

```
#include <functional>
#include <algorithm>
```

The Machine that Goes "Ping"

The Machine that Goes "Ping"



The Machine that Goes "Ping"

```
template <typename It, typename Predicate>
std::pair<It, It>
move_selection(It first, It last,
               It destination,
               Pred predicate)
{
    return std::make_pair(
        std::stable_partition(first, destination,
                              negate(predicate));
        std::stable_partition(destination, last,
                              predicate)
    );
}
```

The Machine that Goes "Ping"

`operator()` + templates

The Machine that Goes "Ping"

```
std::multiplies<float>()
std::bind1st(…)
```

`arg1 * arg2` | boost.phoenix

The Machine that Goes "Ping"

```
std::multiplies<>()
std::bind(…)

arg1 * arg2
```

boost.phoenix

The Machine Stopped

Problems:

- One-off functions
- Problems with iterators
- Problem with tuples

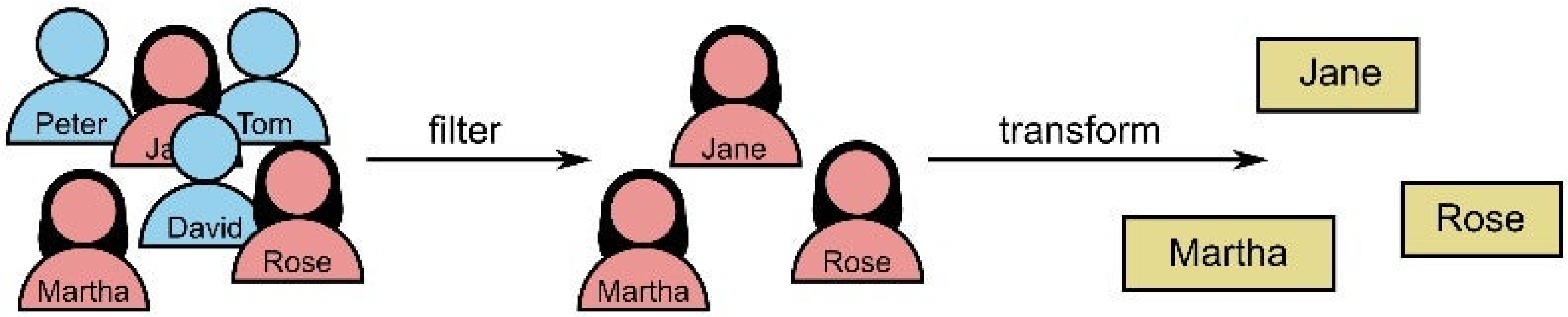
GROWTH AND LEARNING

Lambdas

Lambdas – syntactic sugar for creating function objects:

```
std::stable_partition(
    first, destination,
    [&] (auto&& value) {
        return !predicate(FWD(value));
    });

// FWD - std::forward<decltype(value)>(value)
```

Algorithm composition



Algorithm composition

```
people.erase(
    std::remove_if(people.begin(), people.end(),
                   is_not_female),
    people.end());

// or std::copy_if

std::transform(people.begin(), people.end(),
               std::back_inserer(names),
               &person::name);
```

Algorithm composition

Ranges to the rescue:

```
people | filter(is_female)
       | transform(&person::name);
```

Algorithm composition

```
auto mistery_function(vector<gadget> gadgets, int offset, int count)
{
    vector<gadget> result;
    int skipped = 0, took = 0;

    for (const auto &gadget: gadgets) {
        if (is_container(gadget))
            continue;

        vector<gadget> children;

        for (const auto &child: children(gadget)) {
            if (is_visible(child)) {
                if (skipped < offset) {
                    skipped++;
                } else if (took <= count) {
                    took++;
                    children.push_back(child);
                }
            }
        }

        copy(children.cbegin(), children.cend(), back_inserter(result));
    }

    return result;
}
```

Algorithm composition

```
auto mistery_function(vector<gadget> gadgets,
                      int offset, int count)
{
    return gadgets | filtered(is_container)
                   | transformed(children)
                   | flatten()
                   | filtered(is_visible)
                   | drop(offset)
                   | take(count);
}
```

The Meaning of Tuples

```
template <typename It, typename Predicate>
std::pair<It, It>
move_selection(It first, It last,
               It destination,
               Pred predicate)
{
    …
}
```

The Meaning of Tuples

```
auto selection = move_selection(…);

std::sort(selection.first, selection.second);
```

The Meaning of Tuples

```
auto [ selected_begin, selected_end ] =
    move_selection(…);

std::sort(selected_begin, selected_end);
```

FIGHTING EACH OTHER

Word counting

Problem:
Create a program that fetches a web page
and counts the words in that page

Word counting

States:

- The initial state
- The counting state
- The final state

Word counting

```
struct state_t {
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    string url;
    socket_t web_page;
};
```

not needed until started

Word counting

```
struct state_t {
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    string url;
    socket_t web_page;
};
```

not needed after started

Word counting

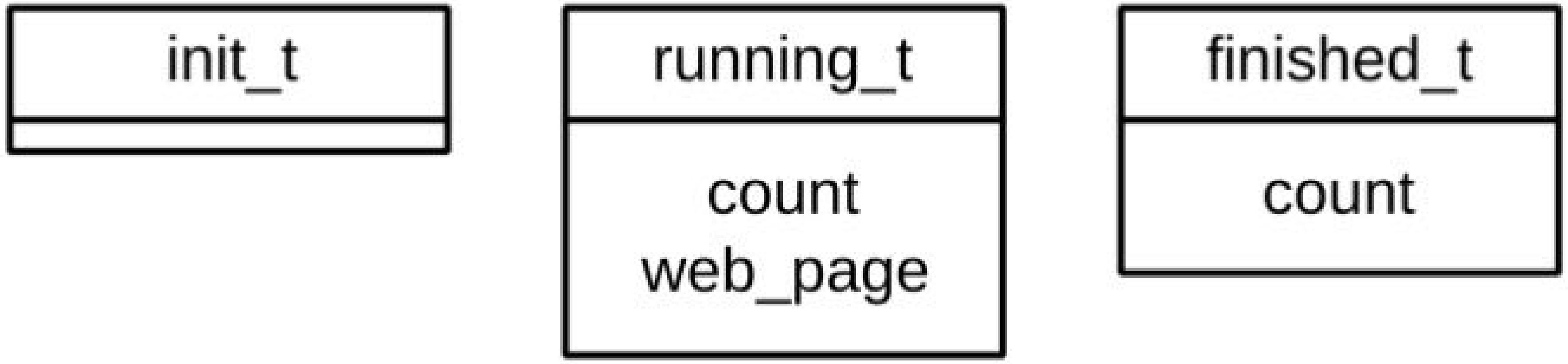
```
struct state_t {
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    string url;
    socket_t web_page;
};
```

needed only while working

Word counting

```
struct state_t {
    bool started = false;
    bool finished = true;
    unsigned count = 42;
    string url = ”http://isocpp.org”;
    socket_t web_page = …;
};
```

Fighting each other



Sum types

A sum of types A and B is a type that contains an instance of A **or** an instance of B, but **not both** at the same time.

$A \cup B$

Sum types using inheritance

We can implement sum types through inheritance.

Create a `state_t` super-class, and sub-classes for each of the states.

Sum types using inheritance

```
class state_t {
protected:
    state_t(int type)
        : type(type)
    {
    }

public:
    const int type;
    virtual ~state_t() {};
};
```

we can not create instances of this type directly

Sum types using inheritance

```
class init_t: public state_t {
public:
    enum { id = 0 };

    init_t()
        : state_t(id)
    {
    }

    string url;
};
```

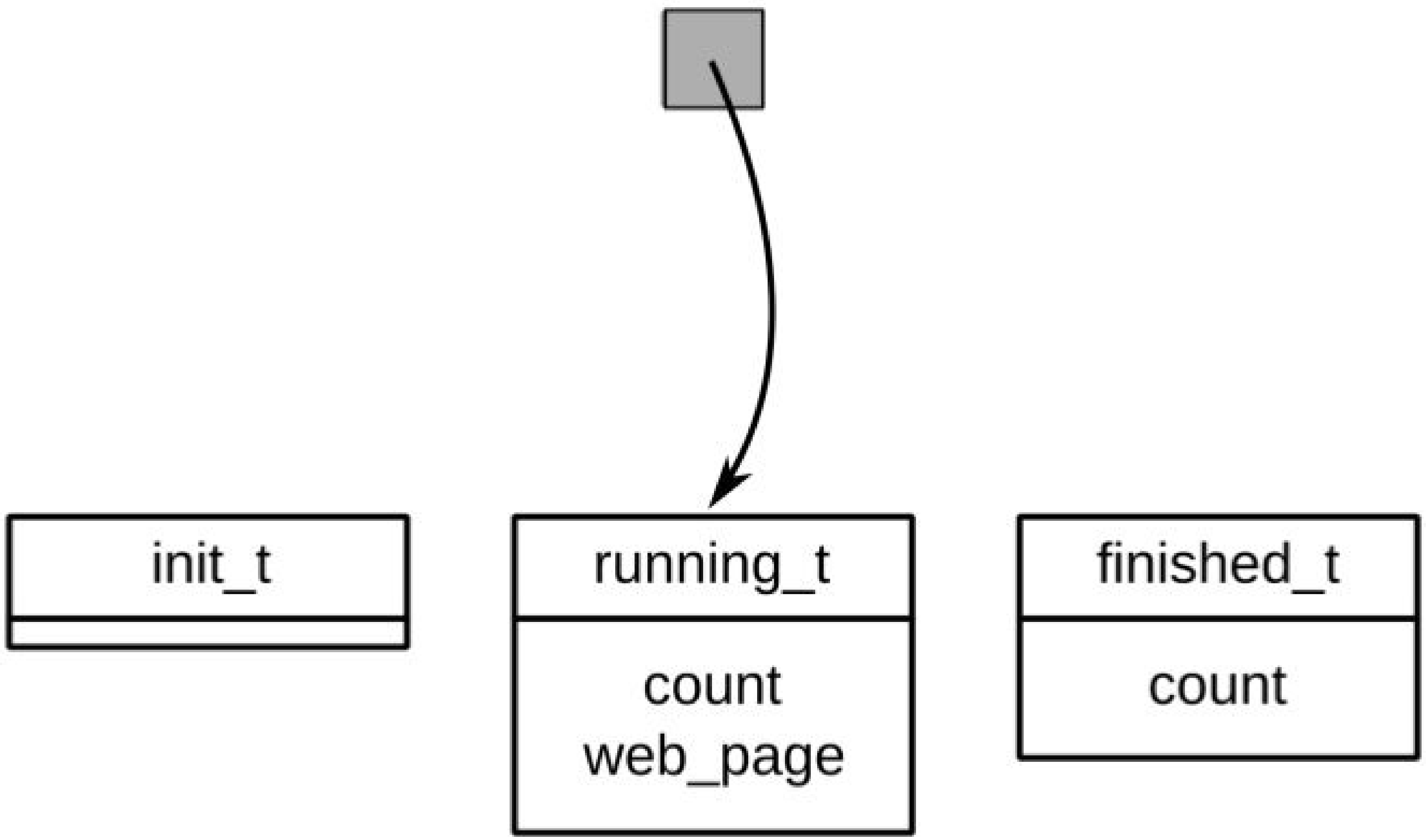
Sum types using inheritance

```
class running_t: public state_t {
public:
    enum { id = 1 };

    init_t()
        : state_t(id)
    {
    }

    unsigned m_count = 0;
    socket_t m_web_page;
};
```

Sum types using inheritance

```
class finished_t: public state_t {
public:
    enum { id = 2 };

    init_t()
        : state_t(id)
    {
    }

    const unsigned m_count = 0;
};
```

Sum types using inheritance



Sum types using inheritance

```
class program_t {
public:
    program_t()
        : m_state(make_unique<init_t>())
    {
    }

    void counting_finished()
    {
        assert(m_state->type == running_t::id);
        auto state =
            static_cast<running_t*>(m_state.get());

        m_state.reset(
            make_unique<finished_t>(state->m_count));
    }

private:
    unique_ptr<state_t> m_state;
};
```

The initial program state

Sum types using inheritance

```
class program_t {
public:
    program_t()
        : m_state(make_unique<init_t>())
    {
    }

    void counting_finished()
    {
        assert(m_state->type == running_t::id);
        auto state =
            static_cast<running_t*>(m_state.get());

        m_state.reset(
            make_unique<finished_t>(state->m_count));
    }

private:
    unique_ptr<state_t> m_state;
};
```

We must have been running in order for this function to be called

Sum types using inheritance

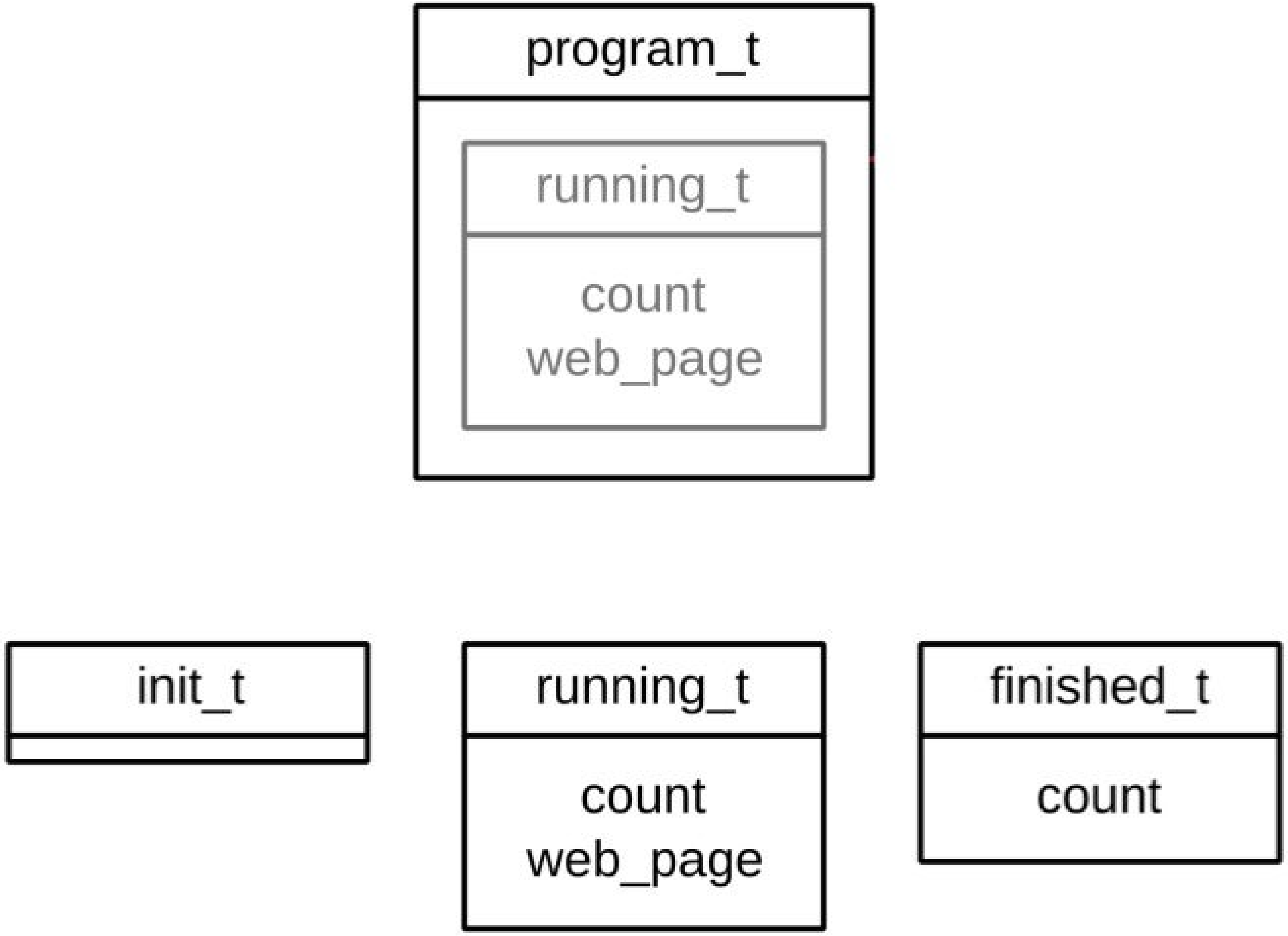
```
class program_t {
public:
    program_t()
        : m_state(make_unique<init_t>())
    {
    }

    void counting_finished()
    {
        assert(m_state->type == running_t::id);
        auto state =
            static_cast<running_t*>(m_state.get());

        m_state.reset(
            make_unique<finished_t>(state->m_count));
    }

private:
    unique_ptr<state_t> m_state;
};
```

Changing state

Sum types using inheritance

- We can have no invalid states
- Set of states is easily extendable (open sum types)
- Automatic resource disposal for resources tied to a particular state
- Bad: A lot of boilerplate

Type-safe unions



Type-safe unions

```
struct init_t { string url; };

struct running_t {
    unsigned m_count;
    socket_t m_web_page;
};

struct finished_t {
    const unsigned m_count;
}

std::variant < init_t
             , running_t
             , finished_t
             > m_state;
```

We can just list all the types that we want to use for state handling

Type-safe unions

```
void counting_finished()
{
    auto *state = get_if<running_t>(&m_state);

    assert(state != nullptr);

    m_state = finished_t(state->m_count);
}
```

Type-safe unions

- We can have no invalid states
- Set of states is fixed (closed sum types)
- Automatic resource disposal for resources tied to a particular state
- No boilerplate

Type-safe unions

```
std::visit(
    [] (const auto& value) {
        std::cout << value << std::endl;
    }, m_state);
```

Type-safe unions

```
template <typename... Ts>
struct overloaded: Ts... { using Ts::operator()...; };

std::visit(overloaded {
        [&] (const init_t& state) {
            …
        },
        [&] (const running_t& state) {
            …
        },
        [&] (const finished_t& state) {
            …
        }
    }, m_state);
```

Optional values

`std::optional<T>` – a sum type of *T* and nothing

```
struct nothing_t {};

template <typename T>
using optional = variant<nothing_t, T>;
```

But with a nicer API.

Optional values

- When we have optional arguments
- When a function can fail

...

Optional values

```
template <typename T, typename Variant>
optional<T> get_if(const Variant& variant)
{
    T* ptr = std::get_if<T>(&variant);

    if (ptr) {
        return *ptr;
    } else {
        return optional<T>();
    }
}
```

Error handling

The `optional` does not keep information about the error.

```
template<typename T, typename E = std::exception_ptr>
class expected {
public:
    // …

private:
    union {
        T m_value;
        E m_error;
    };

    bool m_valid;
};
```

Error handling

```
template<typename T, typename E = std::exception_ptr>
class expected {
public:
    const T& get() const
    {
        if (!m_valid) {
            throw logic_error("Missing a value");
        }

        return m_value;
    }

    // …

private:
    // …
};
```

Error handling

```
template<typename T, typename E = std::exception_ptr>
class expected {
public:
    template <typename... ConsParams>
    static expected success(ConsParams&& ...params)
    {
        expected result;
        result.m_valid = true;
        new (&result.m_value)
            T(forward<ConsParams>(params)...);
        return result;
    }

    // …

private:
    // …
};
```

Error handling

```
template<typename T, typename E = std::exception_ptr>
class expected {
public:
    ~expected()
    {
        if (m_valid) {
            m_value.~T();
        } else {
            m_error.~E();
        }
    }

    // …

private:
    // …
};
```

Error handling

```
template<typename T, typename E = std::exception_ptr>
class expected {
public:
    expected(const expected& other)
        : m_valid(other.m_valid)
    {
        if (m_valid) {
            new (&m_value) T(other.m_value);
        } else {
            new (&m_error) E(other.m_error);
        }
    }

    // …

private:
    // …
};
```

Error handling

```
public:
    void swap(expected& other)
    {
        using std::swap;
        if (m_valid) {
            if (other.m_valid) {
                swap(m_value, other.m_value);

            } else {
                auto temp = std::move(other.m_error);

                other.m_error.~E();
                new (&other.m_value) T(std::move(m_value));

                m_value.~T();
                new (&m_error) E(std::move(temp));

                std::swap(m_valid, other.m_valid);
            }
        } else {
            // …
```

Error handling

```
public:
    void swap(expected& other)
    {
        using std::swap;
        if (m_valid) {
            // …
        } else {
            if (other.m_valid) {
                other.swap(*this);

            } else {
                swap(m_error, other.m_error);
            }
        }
    }
};
```

Handling optional and expected values

```
optional<string> login = current_user();
if (!user) return error

optional<user_t> user = get_info(login.get());
if (!user) return error

optional<string> full_name =
    get_full_name(user.get());
if (full_name) return error

…
```

Handling optional and expected values

Optional (and expected) values are like containers with at most one element.

Handling optional and expected values

```
current_user: () -> optional<string>
get_info: string -> optional<user_t>
get_full_name: user_t -> optional<string>

current_user()
    | transform(get_info)
    | transform(get_full_name)
    | …
    ;
```

Handling optional and expected values

```
current_user: () -> optional<string>
get_info: string -> optional<user_t>
get_full_name: user_t -> optional<string>

current_user()
    | mbind(get_info)
    | mbind(get_full_name)
    | …
    ;
```

THE AUTUMN YEARS

Handling future values

Future values are like containers that will get the value later.

Handling future values

```
current_user: () -> future<string>
get_info: string -> future<user_t>
get_full_name: user_t -> future<string>

current_user()
        | mbind(get_info)
        | mbind(get_full_name)
        | …
        ;
```

Handling future values

```
current_user: () -> future<string>
get_info: string -> future<user_t>
get_full_name: user_t -> future<string>

current_user()
    .then(get_info)
    .then(get_full_name)
    …
    ;
```

Imperative on top

```
current_user: () -> future<string>
get_info: string -> future<user_t>
get_full_name: user_t -> future<string>

auto user = co_await current_user();
auto info = co_await get_info(user);
auto name = co_await get_full_name(info);
```

Imperative on top

`co_await` expression is equivalent to:

```
{
    auto && tmp = <expr>;
    if (!await_ready(tmp)) {
        await_suspend(tmp, continuation);

    }
    return await_resume(tmp);
}
```

Imperative on top

A core-language feature to implement `.then`

Imperative on top

A core-language feature to implement `mbind`

Imperative on top

```
current_user: () -> optional<string>
get_info: string -> optional<user_t>
get_full_name: user_t -> optional<string>

auto user = co_await current_user();
auto info = co_await get_info(user);
auto name = co_await get_full_name(info);
```

Summary

- First, higher-order functions
- Then, alternative type design safe state handling with sum types
- Combination of the two
- Monadic design
- Finally baking the monad into the core-language

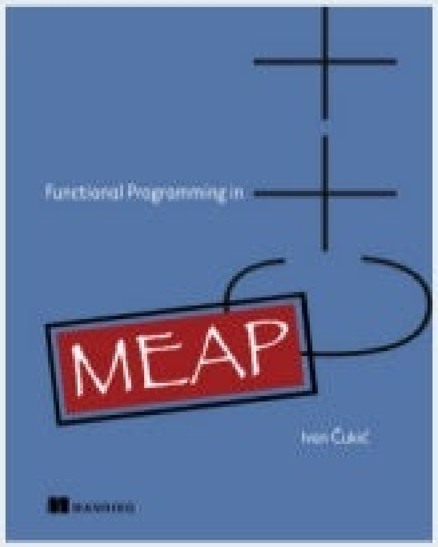
Compilation process

- Write imperative `co_await`-based code
- Converted to functional (monadic) design
- Uses TMP for the type
- Unwraps all into imperative code
- Converted to a single-assignment internal compiler language
- To be finally converted to assembly

Answers? Questions! Questions? Answers!

Kudos (in chronological order):

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



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

Discount code (40% off all Manning books):
ctwcorehard