Monads in Chains

Meeting C++, Berlin 2014

Ivan Čukić

ivan.cukic@kde.org
ivan.fomentgroup.org/blog

About me

- KDE development
- Talks and teaching
- Functional programming enthusiast

Soviet War Memorial
Treptower Park
East Berlin

Soviet War Memorial
Tiergarten
West Berlin

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

Disclaimer

The code snippets are optimized for presentation, it is not production-ready code.

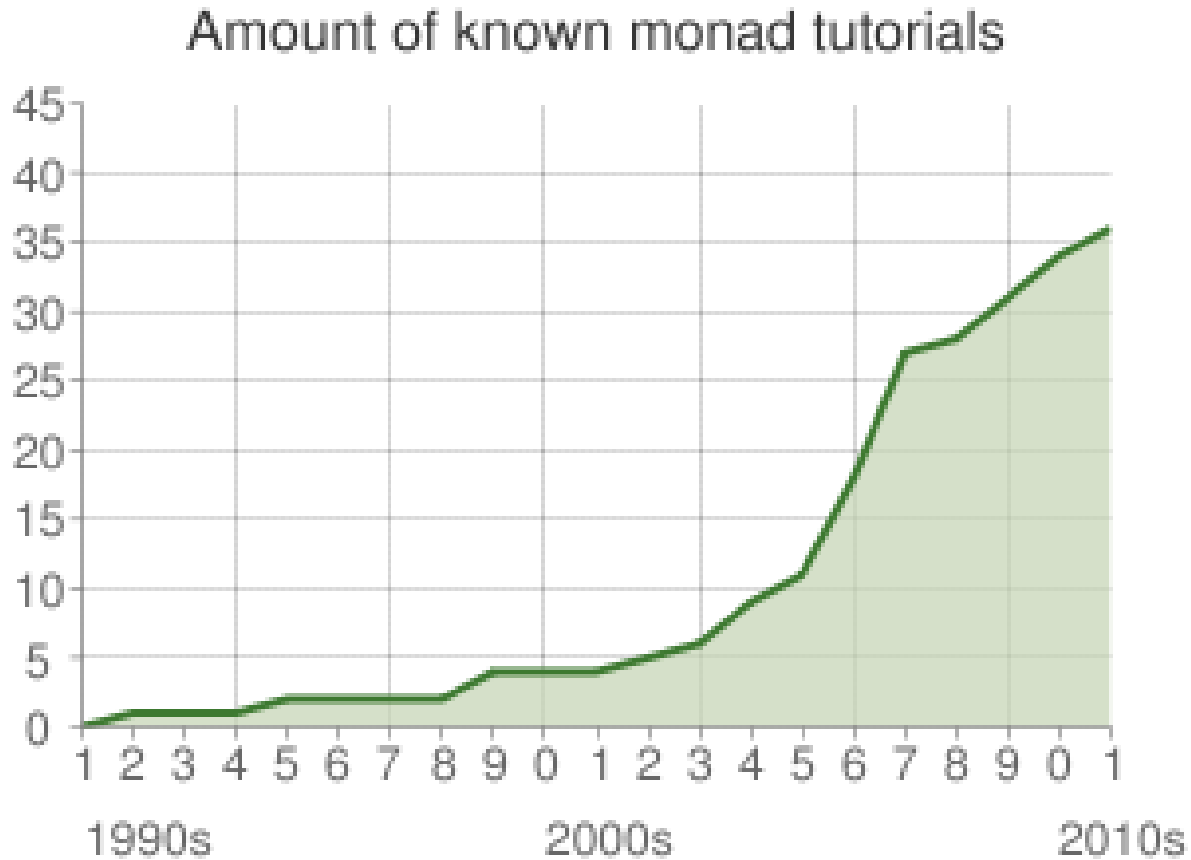
`std` namespace is omitted, value arguments used instead of const-refs or forwarding refs, etc.

Contents

- (Not) Yet another monad tutorial
- Examples of useful monads
- Asynchronous and reactive programming

MONADS

(Not) Yet Another Monad Tutorial



www.haskell.org/haskellwiki/Monad_tutorials_timeline

What is a monad?

A monad is just a monoid in the category of endofunctors, what's the problem?

James Iry as Philip Wadler
(Brief, Incomplete and Mostly Wrong History of Programming Languages)

What is a monad?

All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.

Saunders Mac Lane
(Categories for the Working Mathematician)

Pure functions

Functions that always return the same value when evaluated with the same arguments.

Functions without side-effects. Without (mutable) state.

I/O, random numbers, etc.

```
// (void) -> string
line1 = readLine();
line2 = readLine();

// (int) -> int
x = randLessThan(100);
y = randLessThan(200);
z = randLessThan(100);
```

I/O, random numbers, etc.

```
// (stream) -> (string, stream)
(line1, stream2) = readLine(stream);
(line2, stream3) = readLine(stream2);

// (generator) -> (int, generator)
(x, gen2) = randLessThan(gen,  100);
(y, gen3) = randLessThan(gen2, 100);
(z, gen4) = randLessThan(gen3, 100);
```

I/O, random numbers, etc.

```
// (stream) -> (string, stream)
stream >> word;
std::getline(stream, line);
```

Formal attire

```
-- Monad m
>>=    :: m a -> (a -> m b) -> m b
return :: a -> m a

-- plus a few rules
(return x) >>= f == f x
m >>= return     == m

(m >>= f) >>= g  == m >>= (\x -> (f x >>= g))
```

Formal attire

```
Monad<To> _bind(Monad<From> m,
                function<Monad<To>(From)> f)

Monad<T> _return(T v);
```

EXAMPLES

List

Maybe/Optional

Expected<T>

Other monads

As containers

Future

Reactive streams

List

```
for (item: items) {
    // do something
}

for_each(items, [] (item i) {
    // do something
});
```

List

```
auto get_young_children(list<person> people, int offset, int count)
{
    list<person> result;
    int skipped = 0, took = 0;

    for (person: people) {
        if (is_female(person))
            continue;

        list<person> children;

        for (child: get_children(person)) {
            if (child.age < 20) {
                if (skipped < offset) {
                    skipped++;
                } else {
                    :::
                }
            }
        }

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

    return result;
}
```

List as a monad

```
list<T> _return(T v)
{
    return {v};
}

list<T> _bind(list<F> list,
              function<list<T>(F)> f)
{
    return accumulate(transform(list, f), {});
    //                      ^ list<list<T>>
}
```

List as a monad

```
list<To> transform(list<From> l,
                   function<To(From)> f);

list<T> filter(list<T> l,
               function<bool(T)> predicate);

list<T> flatten(list<list<T>> lists);

list<T> take(list<T> l, int number);

list<T> drop(list<T> l, int number);
```

List, boost.range, N4128

```
auto get_young_children(list<person> people,
                        int offset, int count)
{
    return people | filtered(is_female)
                  | transformed(get_children)
                  | accumulated()
                  | filtered(is_younger_than(20))
                  | drop(offset)
                  | take(count);
}
```

List, boost.range, N4128

N4165 Unified Call Syntax,
N4174 Call syntax: x.f(y) vs. f(x,y)

```
auto get_young_children(list<person> people,
                        int offset, int count)
{
    return people.filtered(is_female)
                 .transformed(get_children)
                 .accumulated()
                 .filtered(is_younger_than(20))
                 .drop(offset)
                 .take(count);
}
```

Maybe, boost.optional, N3690

When and argument or a result value is optional – that is, we could have a specific value, or it can be empty.

Maybe, boost.optional, N3690

```
string config_value(string key);

double sqrt(double value);

void update_record(string key,
                   string value);

int parse_int(string str);
```

Maybe, boost.optional, N3690

```
string config_value(string key,
                    string default_value);
std::pair<double, bool> sqrt(double value);

void update_record(string key,
                   string * value);

int parse_int(string str); // throw on error
```

Maybe, boost.optional, N3690

```
optional<string> config_value(string key);

optional<double> sqrt(double value);

void update_record(string key,
                   optional<string> value);

optional<int> parse_int(string str);
```

Maybe, boost.optional, N3690

```
string get_query_limit() {
    auto config_limit =
         config_value("query_limit");

    if (!config_limit)
        return string();

    auto limit_option = parse_int(config_limit);

    if (!limit_option)
        return string();

    int limit = 1.5 * limit_option.get();

    return " LIMIT " + to_string(limit);
}
```

Maybe, boost.optional, N3690

```
optional<T> _bind(optional<F> opt,
                  function<optional<T>(F)> f)
{
    return opt ? f(opt.get())
               : optional<T>();
}
```

Maybe, boost.optional, N3690

```
string get_query_limit() {
    return (config_value("query_limit")
        | bind(parse_int)
        | transformed([] (int value) {
            return 1.5 * value;
        })
        | transformed([] (int value) {
            return " LIMIT " + to_string(value);
        })
    ).get_value_or("");
}
```

Maybe, boost.optional, N3690

```
string get_query_limit() {
    return (config_value("query_limit")
        | bind(parse_int)
        | transformed(1.5 * _)
        | transformed(to_string)
        | transformed(" LIMIT " + _)
    ).get_value_or("");
}
```

Expected<T>, N4015

Like optional<T>, but with added error info when the value is not present.

Expected<T>, N4015

```
expected<string> config_value(string key);
expected<double> sqrt(double value);

expected<int> parse_int(string str);
```

Expected<T>, N4015

N4015 Variant

```
expected<string, std::exception> config_value(...
expected<double, error_code>     sqrt(...

expected<int, other_error_class> parse_int(...
```

Expected<T>, N4015

```
expected<string> get_query_limit() {
    return config_value("query_limit")
        | bind(parse_int)
        | transformed(1.5 * _1)
        | transformed(to_string)
        | transformed(" LIMIT " + _1)
    ;
}
```

Other monads

- IO
- Parser
- Random number generators

  etc.

Monads as containers

A container-like structure with a few methods defined on it.

- Constructor method that returns a container containing that element

```
(T) -> C<T>
```

- Bind method

```
(C<From>, function<C<To>(From)>) -> C<To>
```

or

- Transform (map) method

```
(C<From>, function<To(From)>) -> C<To>
```

- Flatten method

```
(C<C<T>>) -> C<T>
```

Monads as containers

A container-like structure with a few methods defined on it.

- Constructor method that returns a container containing that element

```
(T) -> C<T>
```

- Bind method

```
(C<From>, function<C<To>(From)>) -> C<To>
```

or

- Transform (map) method

```
(C<From>, function<To(From)>) -> C<To>
```

- Flatten method

```
(C<C<T>>) -> C<T>
```

std::future<T>, boost.future<T>, QFuture<T>

Container for a result of an asynchonous operation.

```
auto futureResult = async(:::);

// Getting the result synchronously
futureResult.get();
```

std::future<T>, boost.future<T>, QFuture<T>

N3558, Boost.Thread $\geq$ 1.55

```
auto futureResult = async(:::);

// Getting the result asynchronously
futureResult.then([] (auto f) {
        // do something with f
        f.get();
    });
```

std::future<T>, boost.future<T>, QFuture<T>

```
template<typename T, typename F>
auto transform(future<T> future, F func) {
    return future.then([] (future<T> future) {
        return func(future.get());
    });
}
```

Reactive streams

- `future  --  value`
- `???     --  list`

Reactive streams

Container for results of a series of asynchonous operations.

- Mouse coordinates
- Client requests in a web server
- Server response chunks

Reactive streams

```
for (item: items) {
    // do something
}

for_each(items, [] (item i) {
    // do something
});
```

Reactive streams

```
// The usual call-callback approach
void on_mouse_move(point mouse) {
    :::
}

gui_mouse_move_func(on_mouse_move);

// Reactive streams
for_each(mouse_movements, on_mouse_move);
```

Reactive streams

```
for_each(
    mouse_movements
        | filtered(scene_contains)
        | transformed(invert_y_coordinate)
    , on_mouse_move);
```

Reactive streams

```
mouse_movements
    | filtered(scene_contains)
    | transformed(invert_y_coordinate)
    | transformed(on_mouse_move);
```

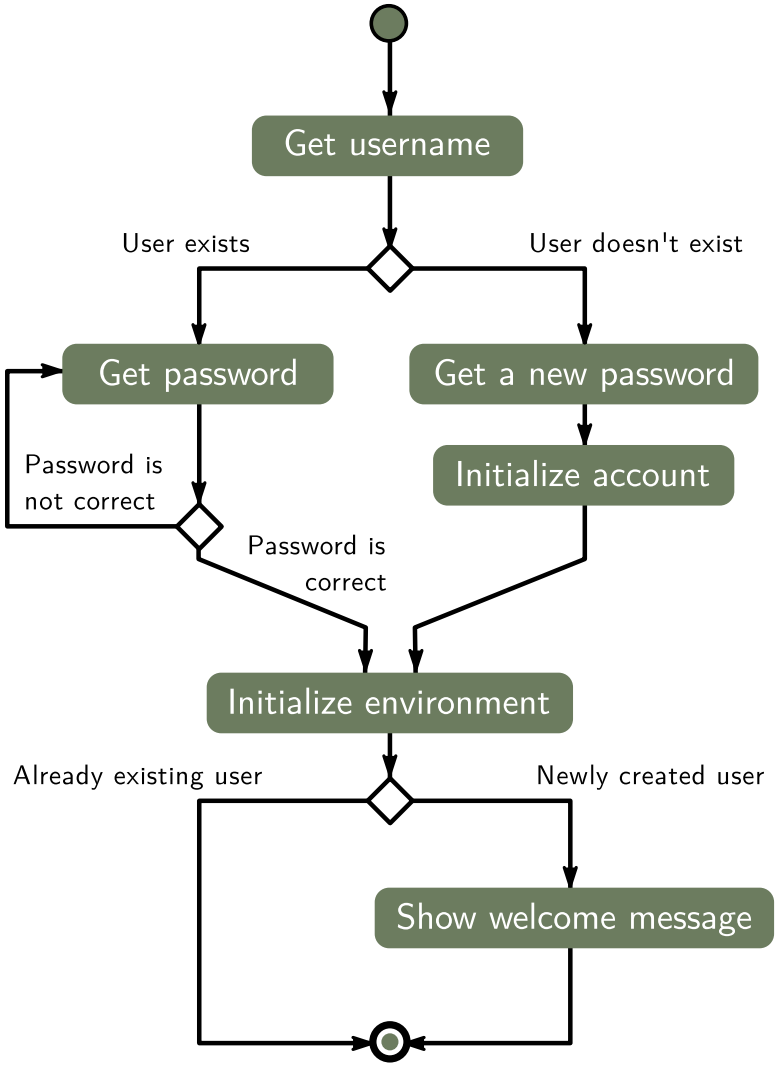
CONTINUATIONS

The problem

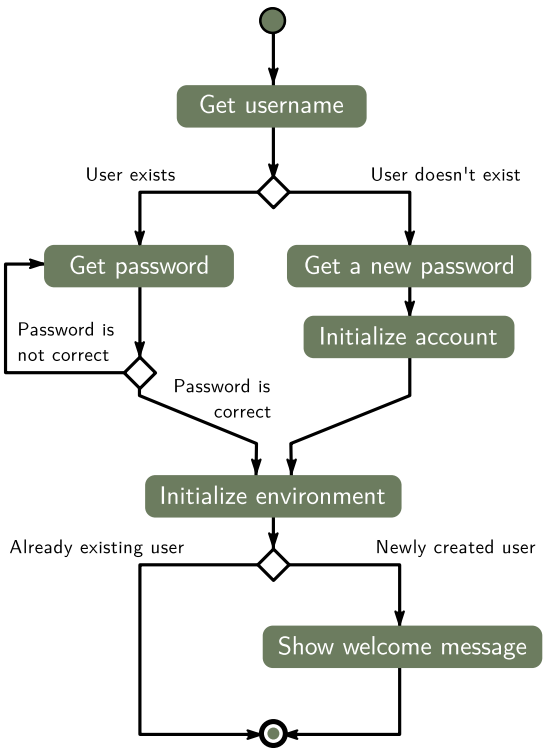
Schedulers

Set Your Controls for the Heart of the Sun

The problem


Get username
User exists
User doesn't exist
Get password
Get a new password
Password is not correct
Initialize account
Password is correct
Initialize environment
Already existing user
Newly created user
Show welcome message

The problem

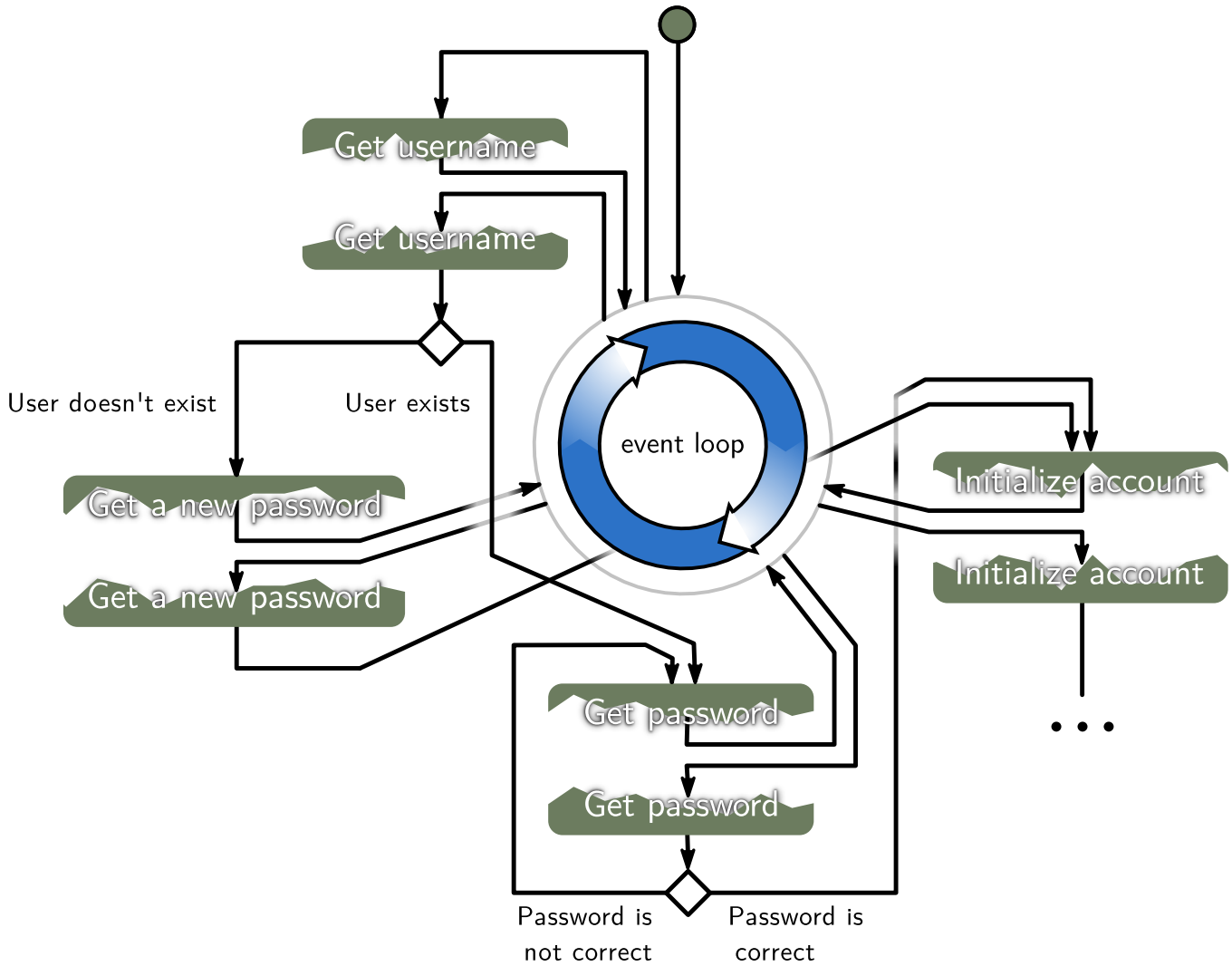



```
void login()
{
    user = get_username();

    new_user = !check_if_user_exists(user);

    if (new_user) {
        pass = get_password();
        initialize_account(uame, pass);

    } else do {
        pass = get_password();

    } while (!check_user(user, pass));

    initialize_environment();

    if (new_user) show_welcome_message();
}
```

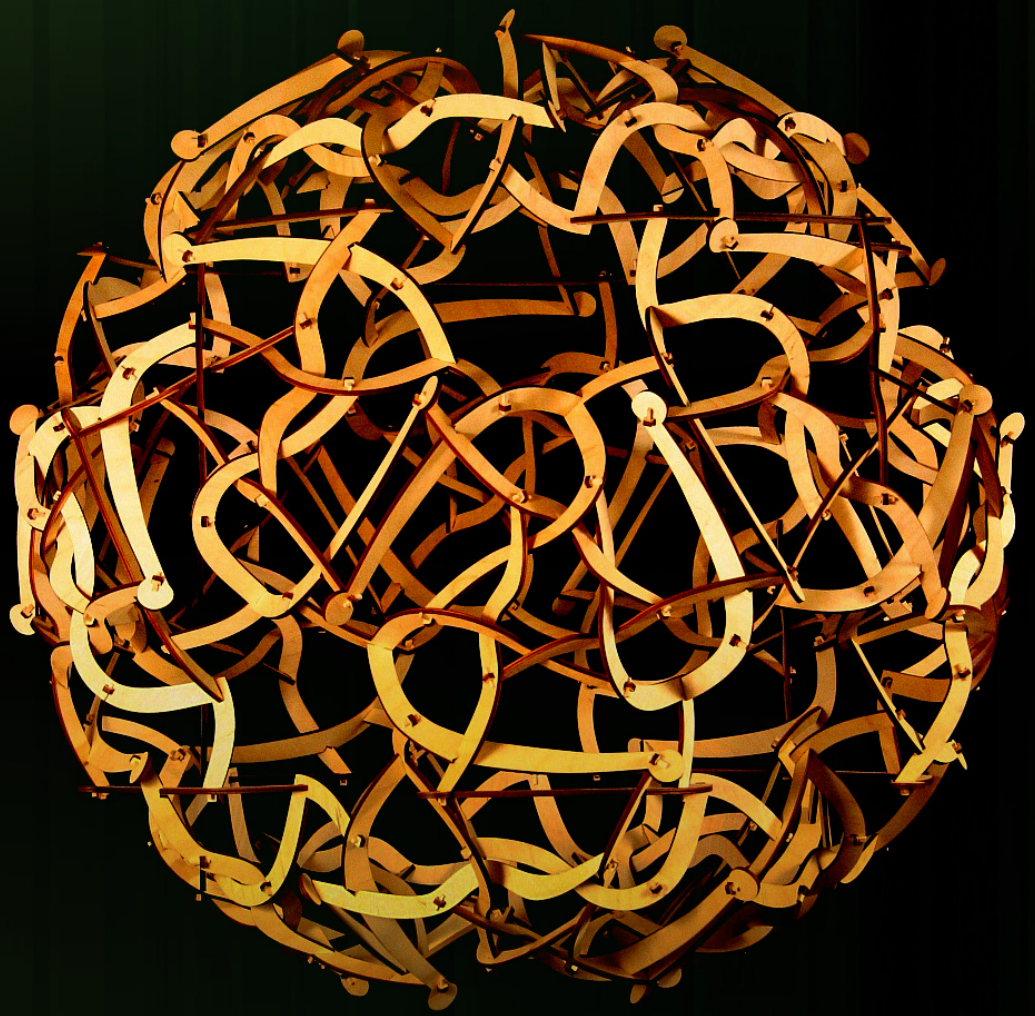
The problem

```
void login() { get_username(on_got_username); }

void on_got_username( ::: ) {
    new_user = !check_if_user_exists(user);
    if (new_user) {
        get_password(on_got_password);
    } else { ::: }
}

void on_got_password( ::: ) {
    check_user(user, password, on_user_checked);
}

void on_user_checked( ::: ) {
    if (!user_valid) {
        on_got_username(user);
    } else {
        initialize_environment(on_environment_initialized);
    }
}

:::
```

Inversion of Control

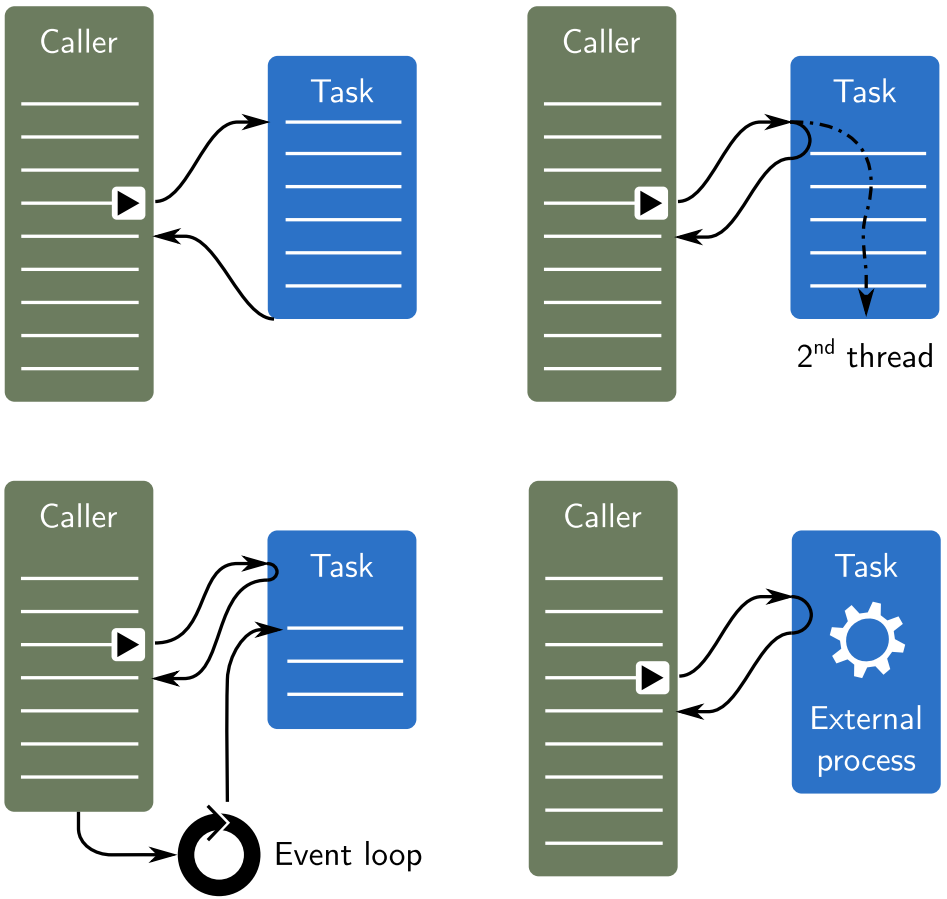


Inversion of Control



"Spaghetti code" by George W. Hart

Reasons for Waiting



- User input
- Network actions
- Inter-process communication
- External process execution
- Communication with a slow database
- CPU-intensive work
- Heterogeneous computing

...

Hiding it all, take 1

- Wrapping it in task objects (`QThread`, `KJob`, ...)
- Methods with time-outs (`select`, ...)
- ... or with validity checks (`QProcess::state`, ...)
- Future values (`future<T>`, `QFuture<T>`, `QDBusPendingReply<T>`, ...)

Hiding it all, take 2

N3558, Boost.Thread $\geq$ 1.55

```
future<int> result = async(:::);

result.then([] (future<int> result) {
    // called when the result is available
    // call to .get() does not block here
    cout << result.get();
});
```

Lost in the Future

```
int i;                          |  i.then(c);          // ERROR!
                                |
future<int> future;             |  future.then(c);     // ok
                                |
QFuture<int> qfuture;           |  qfuture.then(c);    // ERROR!
```

Lost in the Future

```
int i;

future<int> future;

QFuture<int> qfuture;
```

```
c(i);

future.then(c);

auto watcher =
     new QFutureWatcher<int>();
QObject::connect(watcher,
    &QFutureWatcherBase::finished,
    [=] {
        c(watcher->result());
        watcher->deleteLater();
    });
watcher->setFuture(qfuture);
```

Under wraps

```
void continue_with(future<T> f, Function function)

     (future<T>) -> something

     (expected<T>) -> something

     (optional<T>) -> something

     (T) -> something

     () -> something
```

Under wraps

```
template <typename _Future, typename _Continuation>
void continue_with(_Future &&future,
                   _Continuation &&continuation)
{
    using is_nullary =
        typename std::is_constructible<
            std::function<void()>,
            _Continuation
        >::type;

    _continue_with_helper(
        future,
        std::forward<_Continuation>(continuation),
        is_nullary()
    );
}
```

Under wraps

```
template <typename _ReturnType, typename _Continuation>
void _continue_with_helper(const _ReturnType &value,
                           _Continuation &&continuation,
                           std::true_type)
{
    continuation();
}

template <typename _ReturnType, typename _Continuation>
void _continue_with_helper(const _ReturnType &value,
                           _Continuation &&continuation,
                           std::false_type)
{
    using is_callable = ...;
    static_assert(is_callable::value,
        "The continuation needs to at most one argument");

    continuation(value);
}
```

Under wraps

```
template <typename _ReturnType, typename _Continuation>
void _continue_with_helper(const QFuture<_ReturnType> &future,
                           _Continuation &&continuation,
                           std::false_type)
{
    if (!future.isFinished()) {
        auto watcher =
            new QFutureWatcher<_ReturnType>();

        QObject::connect(watcher, &QFutureWatcherBase::finishe
            [=] {
                continuation(watcher->result());
                watcher->deleteLater();
            });

        watcher->setFuture(future);

    } else continuation(future.result());
}
```

Matchbox

```
template<typename _TestType, typename _ArgType>
class has_then_method {
private:
    template<typename U, void (U::*)(_ArgType)>
    struct test_struct {};

    template<typename U>
    static std::true_type test(test_struct <U, &U::then> *);

    template<typename U>
    static std::false_type test(...);

public:
    using type = decltype(test<_TestType>(nullptr));
    static const bool value =
        std::is_same<type, std::true_type>::value;
}
```

The Chains are On

```
getUsername().then(
    [] (future<string> username) {
        getPassword().then(
            [=] (future<string> password) {
                createAccount(username, password).then(
                    ...
                );
            }
        );
    }
);
```

Localized, but still not readable. Can it be made nicer?

The Chains are On

Can it be made to look like this?

```
void login()
{
    ...
    username = getUsername();
    password = getPassword();
    createAccount(username, password);
}
```

No, but ...

The Chains are On

... it could look like this:

```
auto login = serial_
(
    ...
    username = getUsername(),
    password = getPassword(),
    createAccount(username, password)
);
```

Peculiar syntax, but much more readable.

The Chains are On

```
template <typename... _Jobs>
class serial_scheduler;

template <>
class serial_scheduler<> {
public:
    void operator() ()
    {
        on_end_handler(this, EXIT_SUCCESS);
    }

    void on_end(causeway::signal::handler handler) {}
};
```

The Chains are On

```
template <typename _Job, typename... _Jobs>
class serial_scheduler<_Job, _Jobs...> :
    public serial_scheduler<_Jobs...> {
private:
    using tail_t = serial_scheduler<_Jobs...>;
public:
    serial_scheduler(_Job &&job, _Jobs &&... jobs)
        : tail_t(std::forward<_Jobs>(jobs)...)
        , m_job(std::forward<_Job>(job)) {}

    void operator()() {
        auto future = this->future();

        continue_with(std::ref(m_job), [&] {
            tail_t::operator()();
        });

        return future;
    }

private:
    _Job m_job;
};
```

Let There be More Light

- while loop:

```
while_(condition) (
    body
)
```

- branching:

```
if_(condition) (
    then_branch
).else_(
    else_branch
)
```

Let There be More Light

- asynchronous operators

```
var<int> value;

value = 5;             // immediate assignment
value = someFuture(); // asynchronous assignment
```

- parallel execution

```
parallel_(
    task1,
    task2,
    ...
)
```

- parallel without waiting

```
detach_(task)
```

- producer-consumer

```
for_each(clients, process_client);
```

- transactions

etc.

Set Your Controls...

```
var<int> wait;

serial_(
    test::writeMessage(0, "Starting the program"),

    wait = test::howMuchShouldIWait(7),
    test::writeMessageAsync(wait,
            "What is the answer to the Ultimate Question of Life, "
            "the Universe, and Everything?"
        ),

    while_(test::howMuchShouldIWait(0),
        test::writeMessageAsync(1, "42")
    ),

    serial_(
        test::writeMessageAsync(1, "We are going away..."),
        test::writeMessageAsync(1, "... sorry, but we have to.")
    ),

    test::writeMessage(0, "There, you have it!")
)();
```

... for the Heart of the Sun

```
while_(
    // Wait until we get a connection.
    client = ws::server::accept(server),

    // Start a detached execution path to process the client.
    detach_([] {
        var<ws::client_header> header;
        var<ws::message> message;
        var<string> server_key;

        serial_(
            // WebSocket handshake
            header = ws::client::get_header(),
            server_key = ws::server::create_key(header),
            ws::client::send_header(client, server_key),

            // Sending the initial greeting message
            ws::client::message_write(client, "Hello, I'm Echo"),

            // Connection established
            while_(
                // getting and echoing the message
                message = ws::client::message_read(client),
                ws::client::message_write(client, message)
            )
        )
    })
)
```

TASKS

Lazy Day

Lazy Day

Problem:

A method is executed while the arguments are evaluated.

```
serial_(
    someMethod(0, "Starting the program"),
    ...
);
```

`someMethod` must not do anything, but return a functor.

Lazy day

So, your options are:

```
void someMethod(...);

serial_(
    std::bind(someMethod, 0, "Starting the program"
    ...
)
```

Lazy day

Or using a std::bind-based wrapper

```
namespace detail {
    void someMethod(...);
}

auto someMethod(...)
    -> decltype(std::bind(detail::someMethod,
                std::forward arguments ...))
{
    return std::bind(detail::someMethod,
                     std::forward arguments ...);
}

serial_(
    someMethod(0, "Starting the program"),
```

Lazy day

Or using a simple wrapper:

```
namespace detail {
    void someMethod(...);
}

auto someMethod = curry(detail::someMethod);

serial_(
    someMethod(0, "Starting the program"),
    ...
)
```

Answers? Questions! Questions? Answers!

Kudos:

Friends at KDE, Dr Saša Malkov, basysKom

Further reading and watching:

- Iterators Must Go, Andrei Alexandrescu
- Value Semantics and Range Algorithms, Chandler Carruth
- Systematic Error Handling in C++, Andrei Alexandrescu
- Category Theory for Programmers, Bartosz Milewski
  (expected to be awesome when released)
- Learn You a Haskell for Great Good!, Miran Lipovača
  (learning Haskell is fun and can be useful)