Reactive functional and imperative C++

Ericsson, Budapest 2015

Ivan Čukić

KDE
ivan.cukic@kde.org

University of Belgrade
ivan@math.rs

About me

- KDE development
- Talks and teaching
- 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

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.


C++
I AM YOUR FATHER
JS

Why C++

Jun 2015	Jun 2014	Change	Programming Language	Ratings	Change
1	2	▲	Java	17.822%	+1.71%
2	1	▼	C	16.788%	+0.60%
3	**4**	▲	**C++**	**7.756%**	**+1.33%**
4	5	▲	C#	5.056%	+1.11%
5	3	▼	Objective-C	4.339%	-6.60%
6	8	▲	Python	3.999%	+1.29%
7	10	▲	Visual Basic .NET	3.168%	+1.25%
[illegible]					

FUTURES

Concurrency

Futures

Concurrency

- Threads
- Multiple processes
- Distributed systems

Note: "concurrency" will mean that different tasks are executed at overlapping times.

Plain threads are bad

A large fraction of the flaws in software development are due to programmers not fully understanding all the **possible states** their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are **greatly amplified**, almost to the point of panic if you are paying attention.

John Carmack
In-depth: Functional programming in C++

Plain threads are bad

Threads are not composable

Parallelism can't be 'disabled'

Difficult to ensure balanced load manually

Hartmut Kaiser
Plain Threads are the GOTO of Today's Computing

Plain synchronization primitives are bad

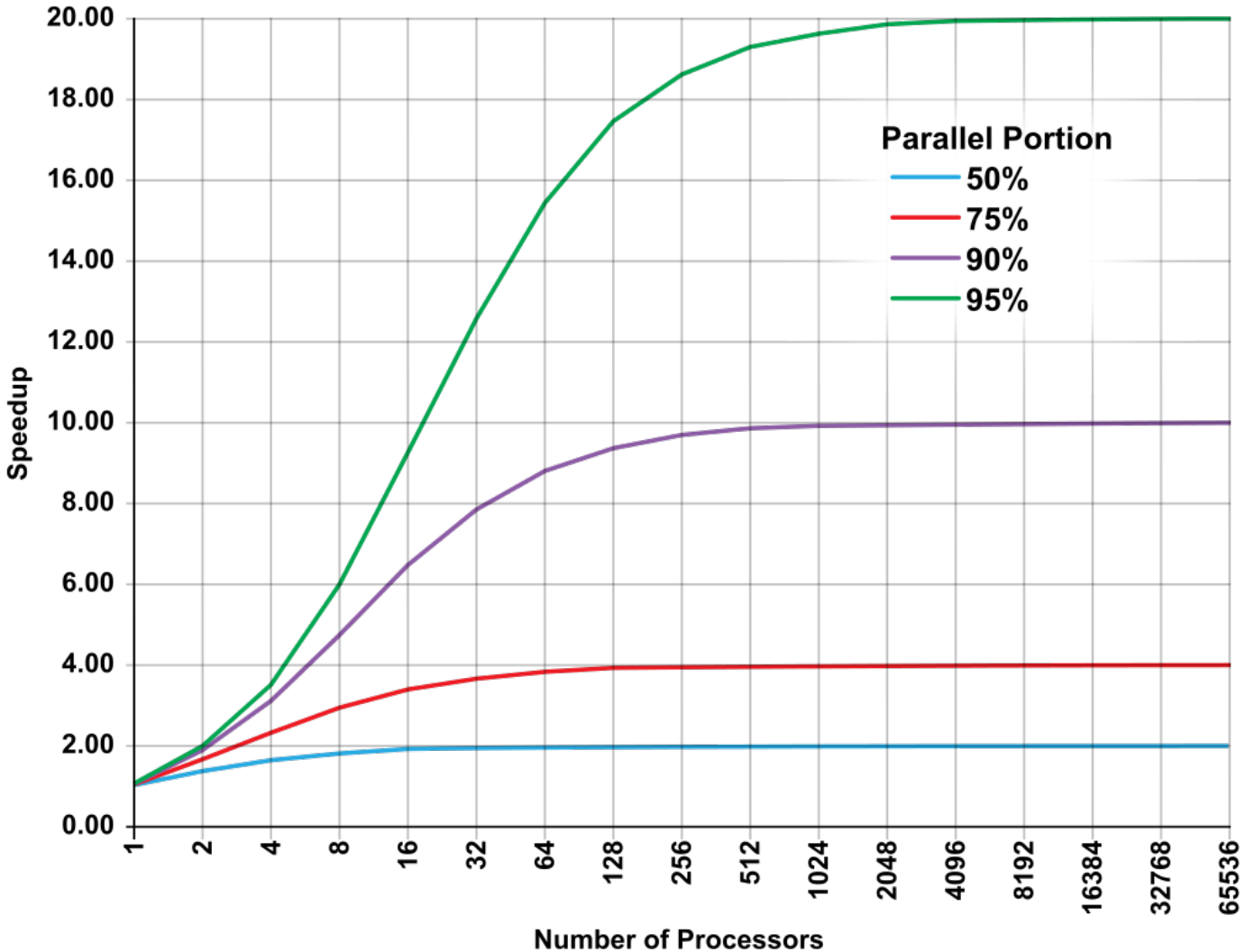
You will likely get it wrong

S.L.O.W. (starvation, latency, overhead, wait)

Sean Parent
Better Code: Concurrency

Amdahl's Law

$$\frac{1}{(1-P)+\frac{P}{N}}$$



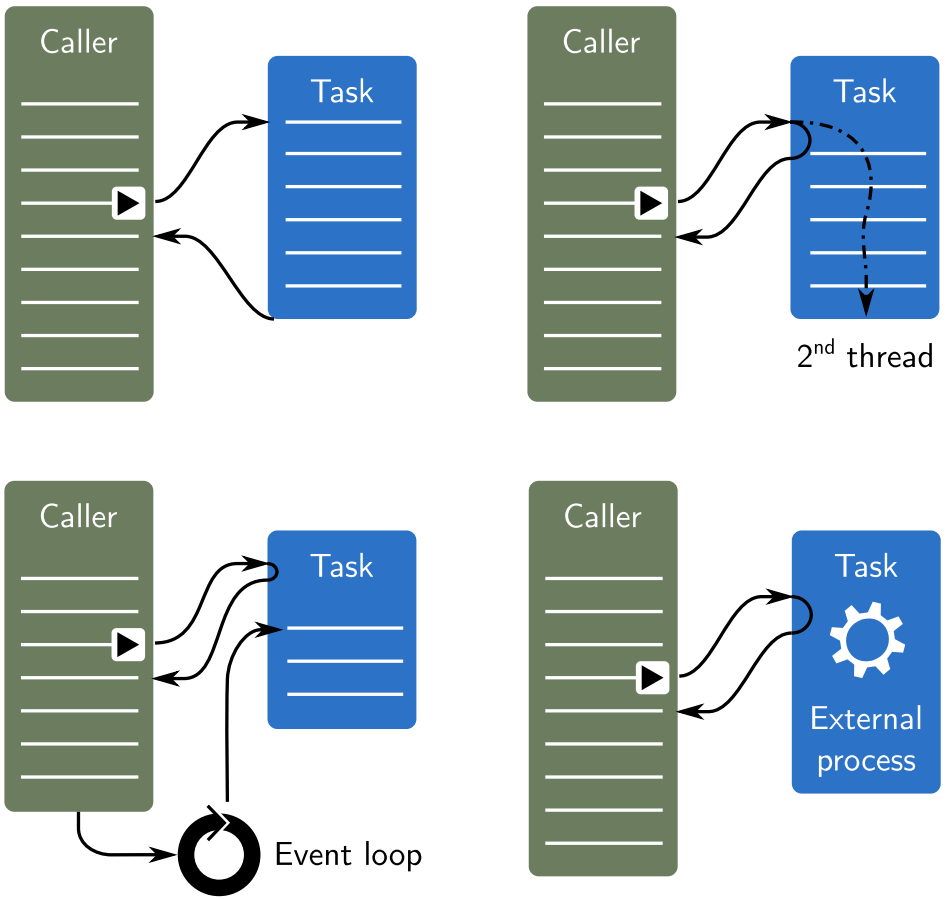
Locks are the main problem

The biggest of all the big problems with recursive mutexes is that they encourage you to completely lose track of your locking scheme and scope. This is deadly. Evil. It's the "thread eater". You hold locks for the absolutely shortest possible time. Period. Always. If you're calling something with a lock held simply because you don't know it's held, or because you don't know whether the callee needs the mutex, then you're holding it too long. You're aiming a shotgun at your application and pulling the trigger. You presumably started using threads to get concurrency; but you've just PREVENTED concurrency.

I've often joked that instead of picking up Djikstra's cute acronym we **should have called the basic synchronization object "the bottleneck"**. Bottlenecks are useful at times, sometimes indispensible – but they're never GOOD.

David Butenhof
Re: recursive mutexes

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

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

Futures

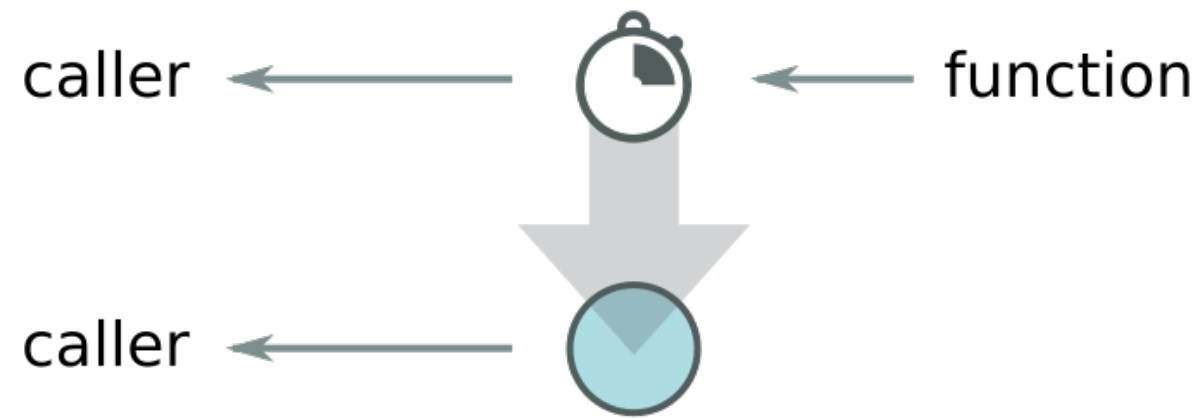
Futures should be the lowest level concurrency abstractions.

```
std::future
boost::future
QFuture
Folly Future

any continuation - *.then([] ...)
```

Future

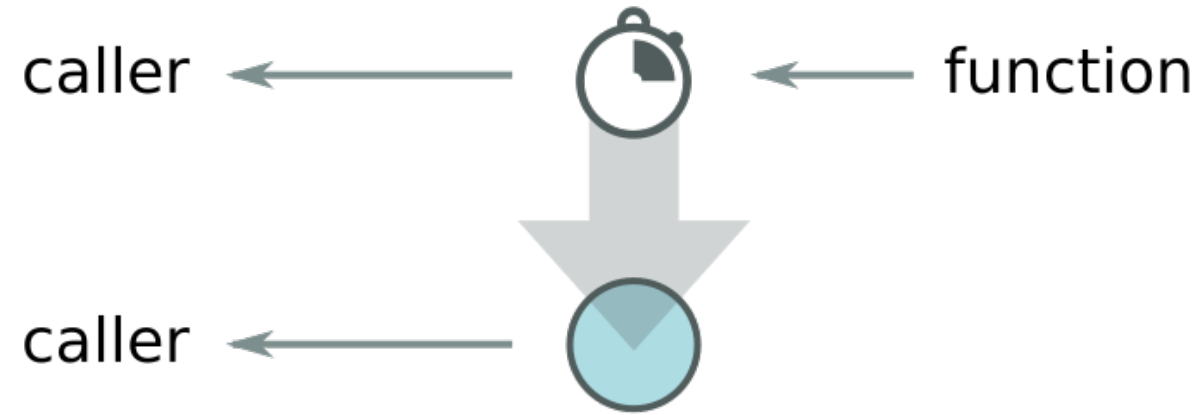
```
T value = function();
```


```
future<T> value = function(); ...; value.get();)
```

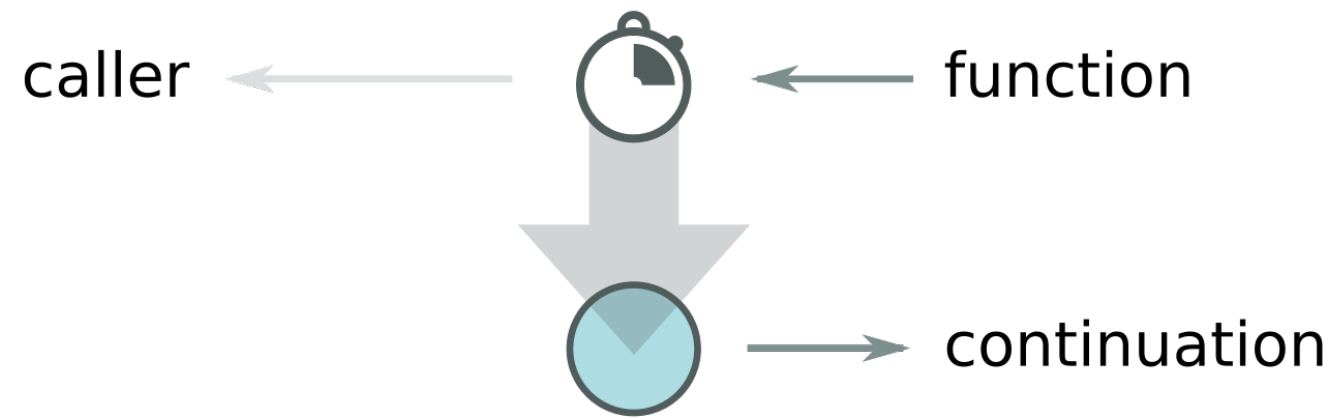


Future

```
future<T> value = function(); ...; value.get();
```




```
future<T2> value = function().then(continuation);
```



Futures

```
get_page("http://people.inf.elte.hu/cefp/")
.then(
    [] (auto result) {
        cout << result.headers();
    }
)
```

Futures

```
get("http://people.inf.elte.hu/cefp/")
.then(
    [] (auto result) {
        cout << result.headers();

        for (image: result.image_tags) {
            image.get().then(
                [] (auto image_result) {
                    // do something
                    // with image_result.
                    // If it needs to be
                    // forwarded, auto&&
                }
            );
        }
    }
```

THE FUNCTIONAL SIDE

STL algorithms

Ranges

Ranges in C++

```
vector<int> xs;
int sum = 0;

for (x: xs) {
    sum += x;
}

return sum;
```

Ranges in C++

```
return accumulate(
    xs.cbegin(), xs.cend(),
    0
    );
```

Ranges in C++

```
return accumulate(
    xs.cbegin(), xs.cend(),
    1,
    _1 * _2
    );
```

Ranges in C++

How to do an aggregation on a transformed list?

```
vector<int> xs;
int sum = 0;

for (x: xs) {
    sum += x * x;
}

return sum;
```

Ranges in C++

How to do an aggregation on a transformed list?

```
sum $ map (λ x → x * x) xs
```

Ranges in C++

How to do an aggregation on a transformed list?

```
vector<int> temp;

std::transform(
    xs.cbegin(), xs.cend(),
    std::back_inserter(temp),
    _1 * _1
    );

return std::accumulate(
    temp.cbegin(),
    temp.cend()
    );
```

Ranges in C++, boost.range, N4128

How to do an aggregation on a transformed list?

```
return accumulate(xs | transformed(_1 * _1)
```

Example

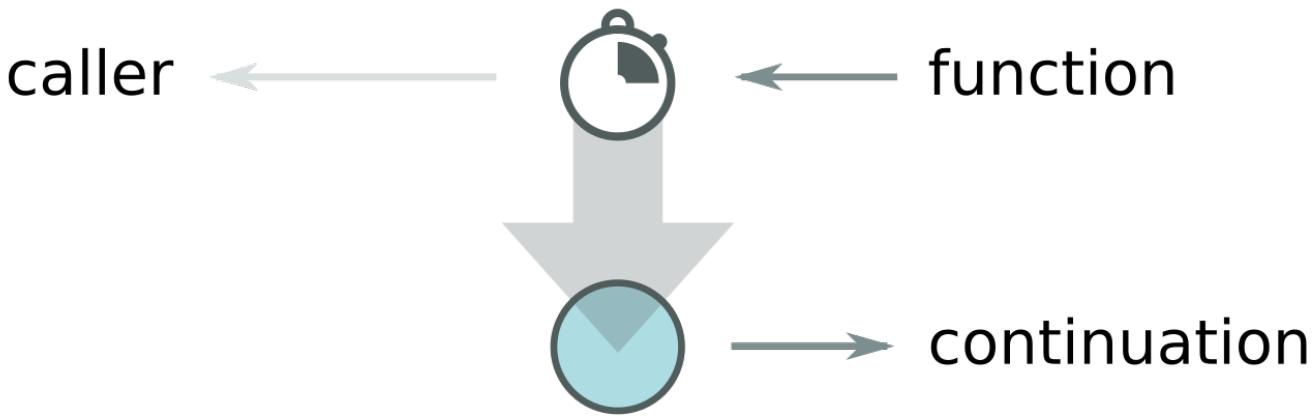
```
transactions
     | filter(Transactions::price() > 1000)
     | groupBy(Transactions::customerId())
     | sort(
          Transactions::price().desc() |
          Transactions::customerName()
       );
```

Example boilerplate

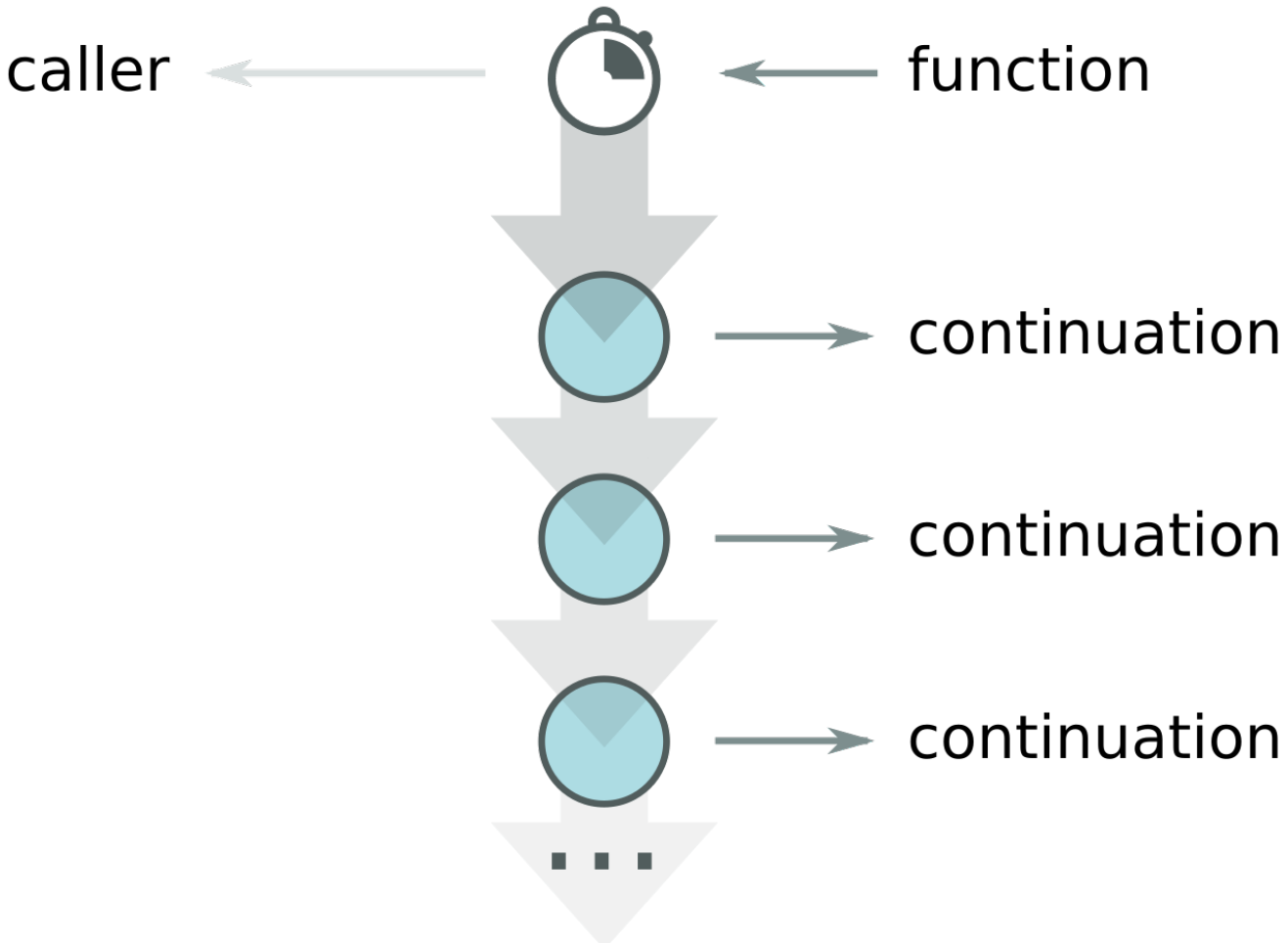
```
namespace Transactions {
    struct Record {
        int customerId;
        ...
    };

    DECL_COLUMN(customerId)
    ...
}
```

Column meta-type has all operators implemented, asc(), desc(), etc.

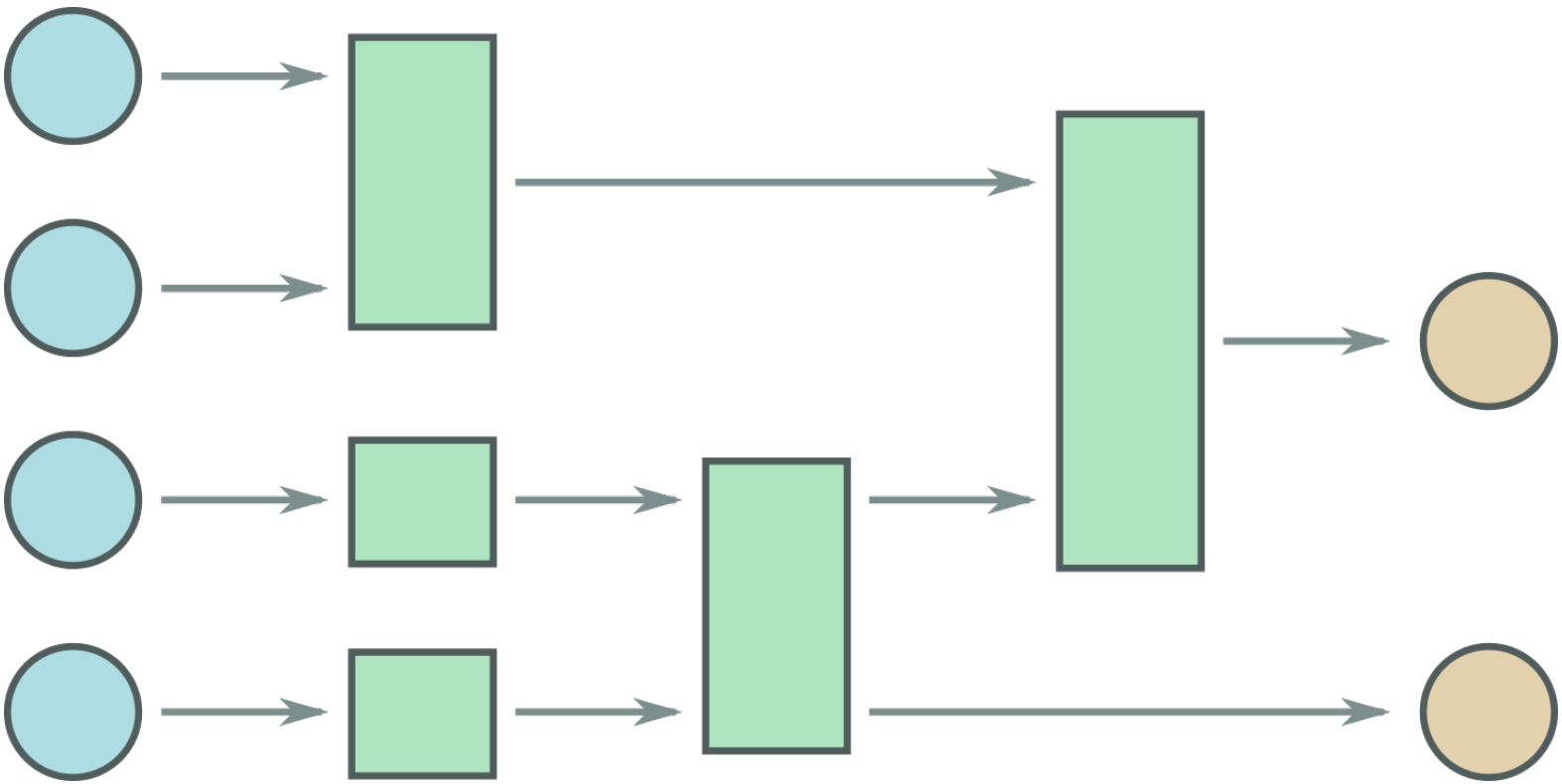
Just passing our time



Oh we'll keep on trying



Flow of information



Through the eons, and on and on

- Web server client connection requests
- User interface events
- Database access
- I/O
- Anything and everything

Till the end of time

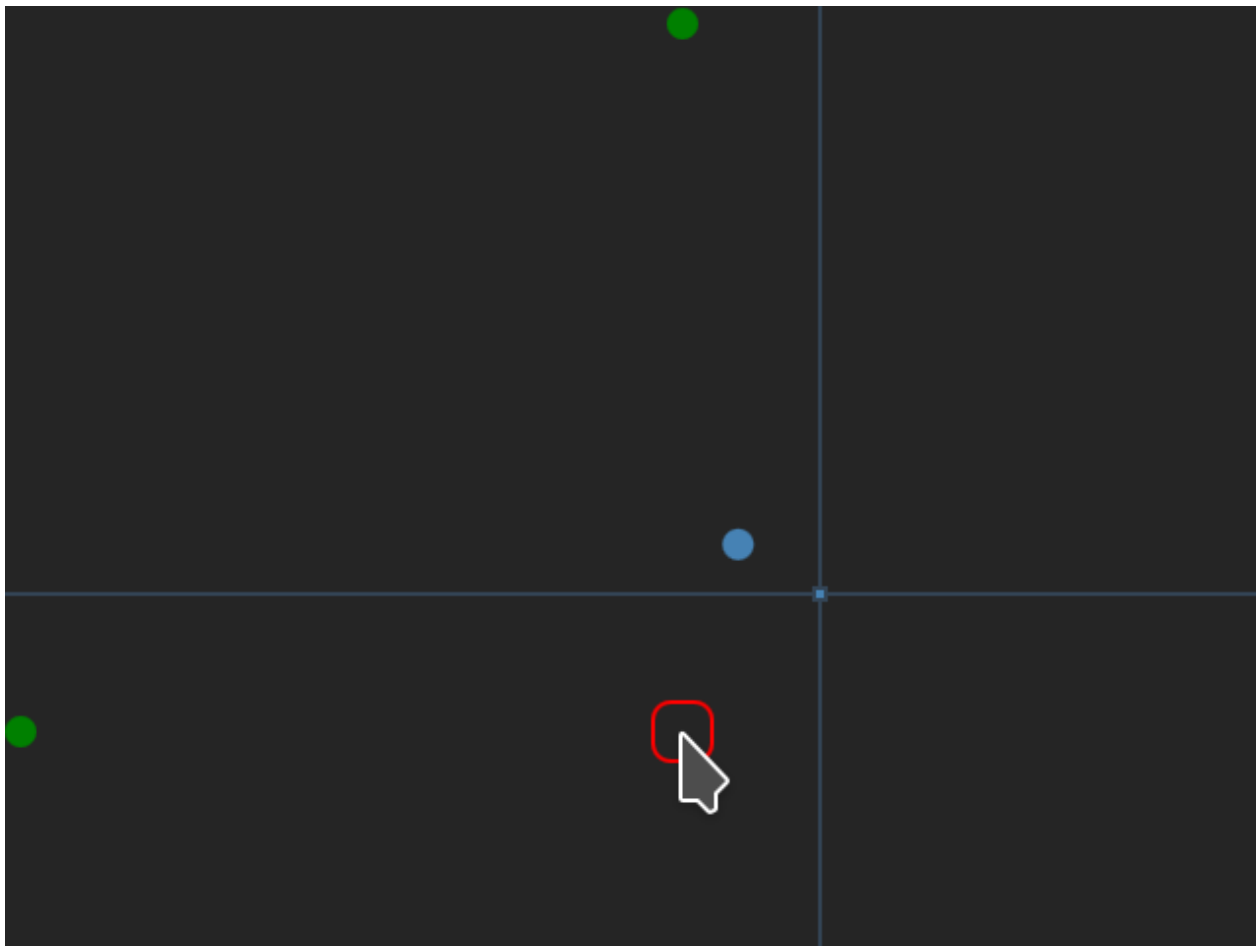
- Message passing:
  `continuation!newClientMessage`
- Call-callback:
  `onNewMessage(continuation)`
- Signals-slots:
```
connect(socket,   &Socket::newConnection,
        receiver, &Receiver::continuation)
```
- Any data collection:
  `for_each(xs, continuation)`

Stream transformation

Streams can only be transformed with algorithms that accept input ranges.

`map`, `bind`, `filter`, `take`, `drop`, etc.

Stream transformation



Map / Transform

We have a stream of 2D coordinates (mouse coordinates).

```
// Projecting on the x-axis
mouse_position >>=
    map(λ point → (point.x, 0))

// Projecting on the y-axis
mouse_position >>=
    map(λ point → (0, point.y))
```

Implementation detail

```
namespace stream {
    template <typename Stream, typename Con
    auto operator >>= (Stream &&stream,
                       Cont &&cont)
    {
        return stream.then(cont);
    }

    template <typename Under>
    auto make_stream(Under &&under);
}
```

Map

```
template <typename Func, typename Cont>
struct map_cont {
    map_cont(Func f, Cont c) : f(f), c(c) { }

            template <typename InType>
            void operator () (const InType &in) {
                c(f(in));
            }

    Func f;
    Cont c;
};
```

Fork (or parallel), tee

```
tee(print) >>=
fork(
    receiver1,
    receiver2
)
```

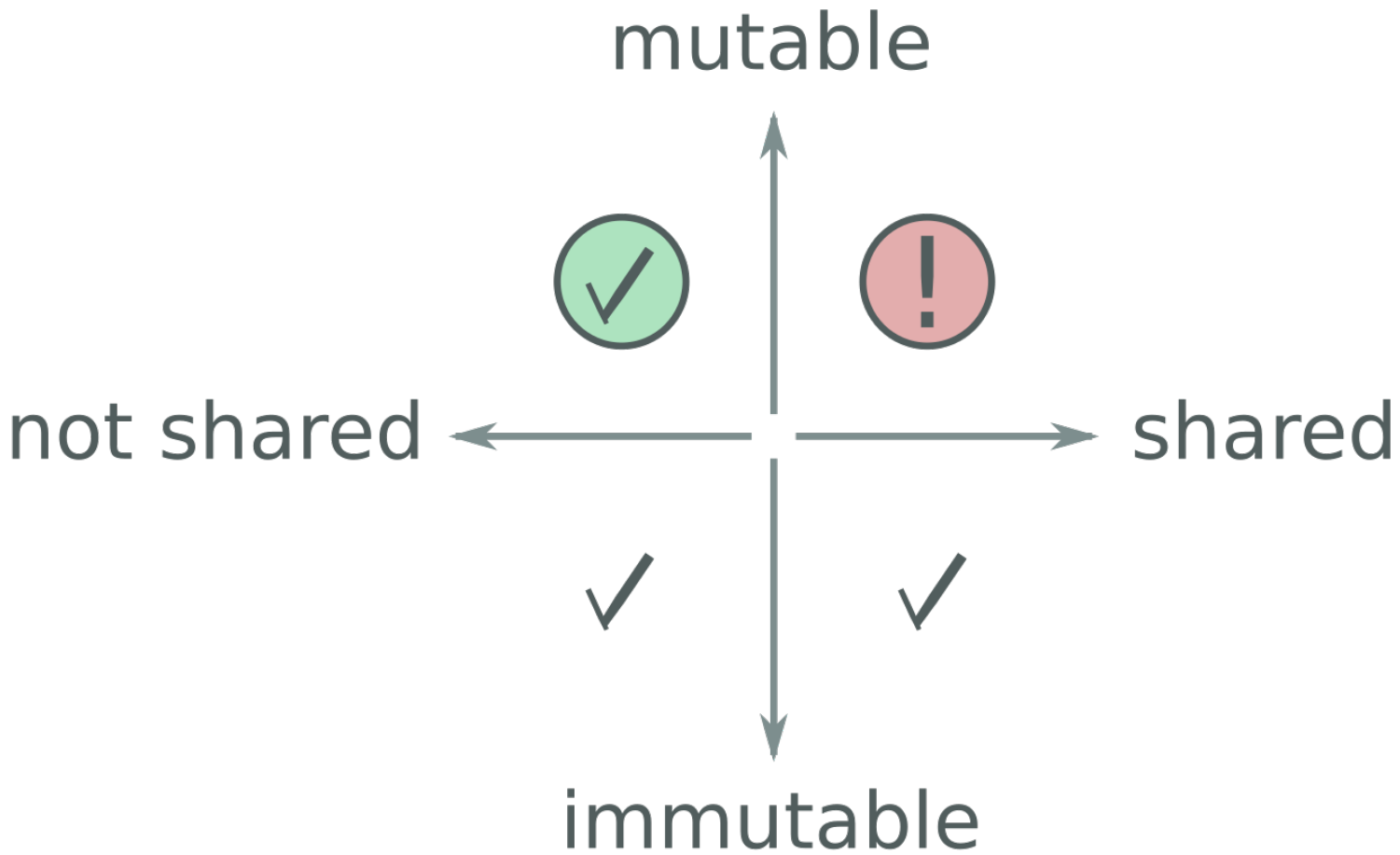
Fork (or parallel), tee

```
template <typename ... Conts>
struct fork_impl;

template <typename Cont, typename ... Conts>
struct fork_impl<Cont, Conts...>: fork_impl<Conts...>
{
    using parent_type = fork_impl<Conts...>;

    fork_impl(Cont c, Conts... cs)
        : parent_type(cs...), c(c)
    { }

            template <typename InType>
            void operator() (const InType &in) {
                c(in);
                parent_type::operator()(in);
            }

    Cont c;
};
```

Stateful function objects

```
class gravity_object {
public:
    gravity_object() { }

    template <typename Cont>
    void then(Cont &&c) { _f = std::forward<Cont>(c); }

            QPointF operator() (const QPointF &new_point) {
                m_point.setX(m_point.x() * .99 + new_point.x() * .01);
                m_point.setY(m_point.y() * .99 + new_point.y() * .01);
                return m_point;
            }

private:
    std::function<void(QPointF)> _f;
    QPointF m_point;
};
```

Stateful function objects



Can stateful function objects be pure?

- Like actors changing behaviour
- Or, treating the function object like its argument is the past part of the stream (a finite list of elements)

Filter

```
bool pointFilter(const QPointF &point) {
    return int(point.y()) % 100 == 0;
}

events >>=
    filter(predicate) >>=
        ...
```

Flat map

```
template <typename Func, typename Cont>
struct flatmap_cont {
    flatmap_cont(Func f, Cont c)
        : f(f)
        , c(c)
    {
    }

        template <typename InType>
        void operator () (const InType &in) {
            boost::for_each(f(in), c);
        }

    Func f;
    Cont c;
};
```

Flat map

```
class more_precision {
public:
    more_precision() { }

    template <typename Cont>
    void then(Cont &&c) { _f = std::forward<Cont>(c); }

            std::vector<QPointF> operator() (const QPointF &new_point) {
                std::vector<QPointF> result;

                int stepX = (m_previous_point.x() < new_point.x()) ? 1 : -1;
                for (int i = (int)m_previous_point.x(); i != (int)new_point.x(); i += stepX)
                    result.emplace_back(i, m_previous_point.y());
                }

                int stepY = (m_previous_point.y() < new_point.y()) ? 1 : -1;
                for (int i = (int)m_previous_point.y(); i != (int)new_point.y(); i += stepY)
                    result.emplace_back(new_point.x(), i);
                }

                m_previous_point = new_point;
                return result;
            }

private:
    std::function<void(QPointF)> _f;
    QPointF m_previous_point;

};
```

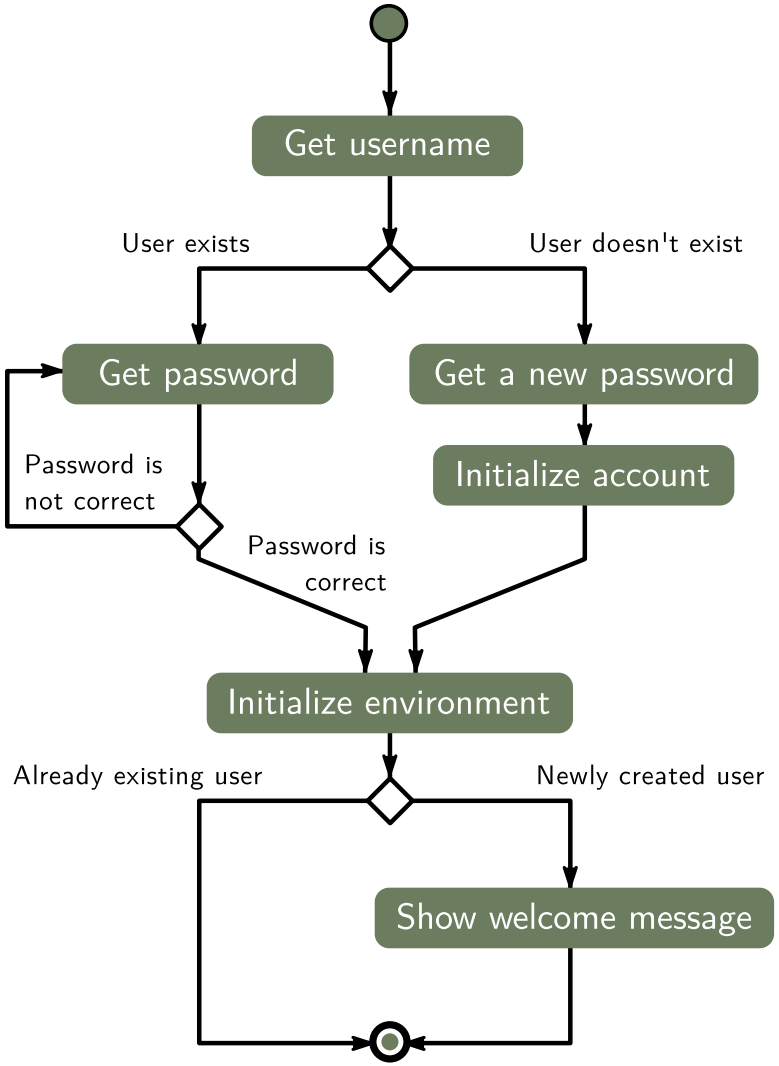
THE IMPERATIVE SIDE

The problem

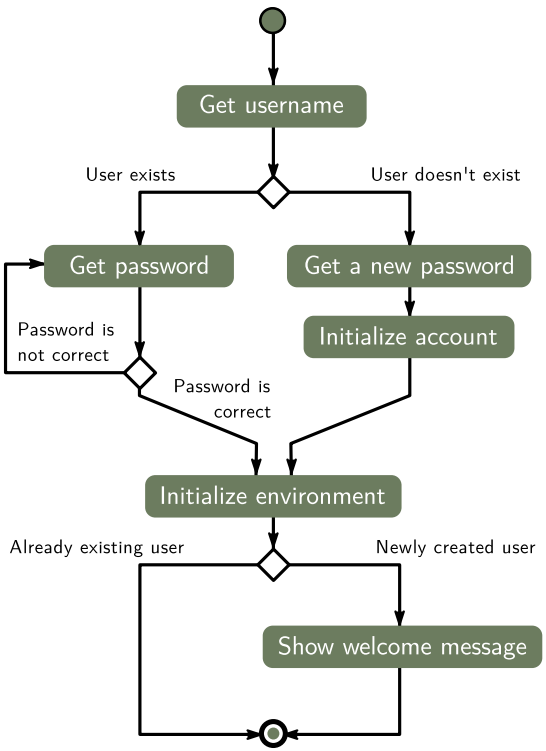
Schedulers

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

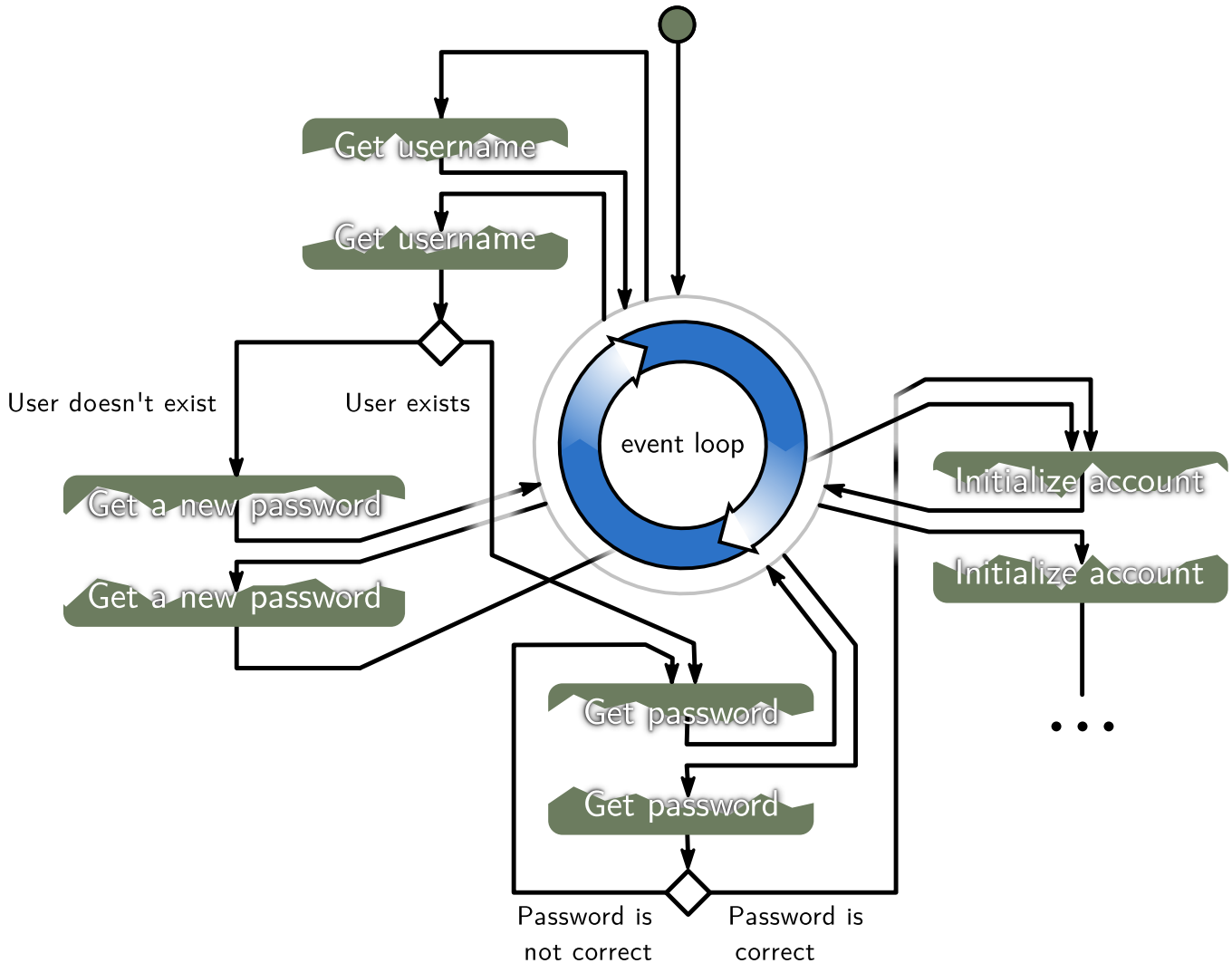



```
void login()
{
    user = get_username();

    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();
}
```

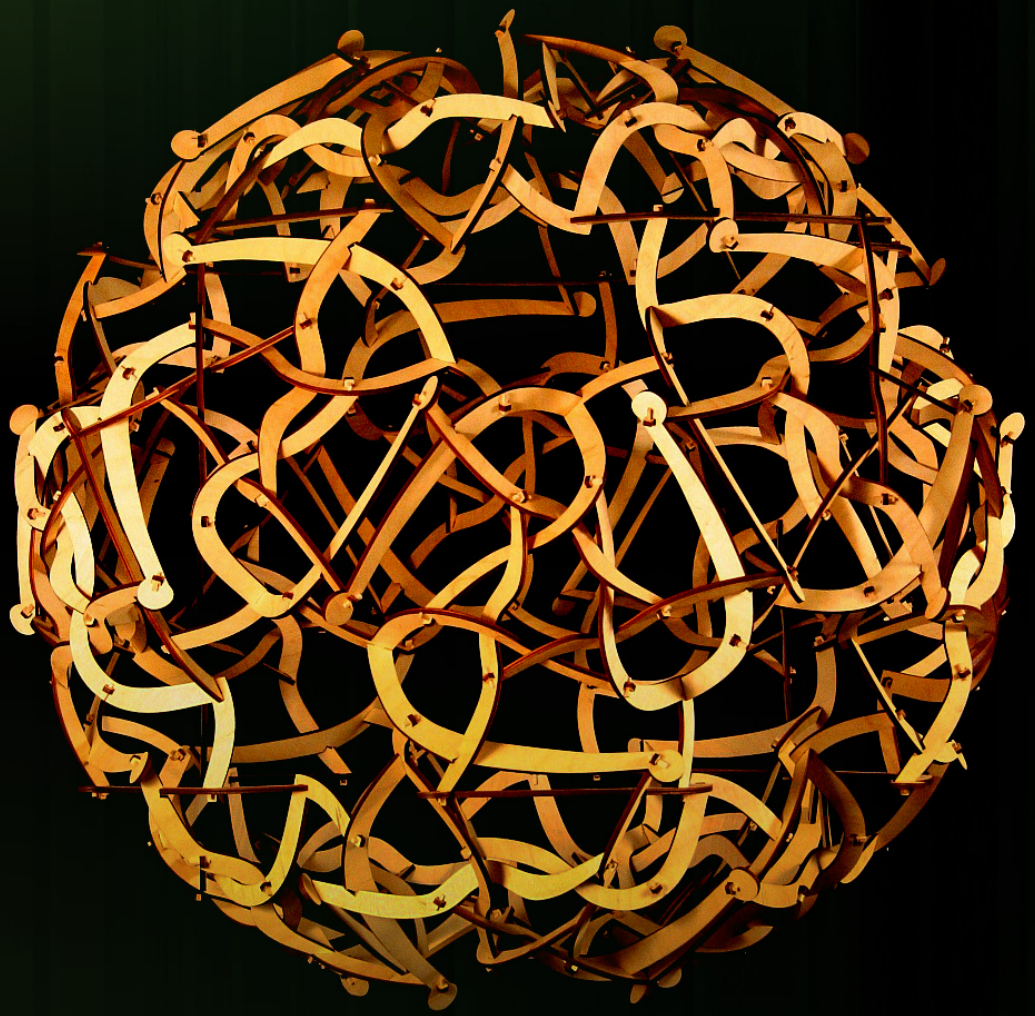
The problem

```
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

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 than the call-callback solution.

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.

Let There be More Light

```
operator(bool) // or start and undo

transaction_(
    task1,
    task2,
    ...
    taskn
    );
```

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 Li
            "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)
            )
        )
    })
)
```

FURTHER EVOLUTION OF C++

Ranges

Await

Ranges

Some call it STL 2.o, provides separate views and actions

Filter a container using a predicate and transform it.

```
std::vector<int> vi{1,2,3,4,5,6,7,8,9,10};
auto rng = vi | view::remove_if([](int i){return i % 2 == 1;})
              | view::transform([](int i){return std::to_string(i);});
```

Generate an infinite list of integers starting at 1, square them, take the first 10, and sum them:

```
int sum = accumulate(view::ints(1)
                   | view::transform([](int i){return i*i;})
                   | view::take(10), 0);
```

Generate a sequence on the fly with a range comprehension and initialize a vector with it:

```
std::vector<int> vi =
    view::for_each(view::ints(1,10), [](int i){
        return yield_from(view::repeat(i,i));
    });
```

Monadic await

```
result = await get(...);

for (image: result.image_tags) (
    image_result = await image.get();
    // do something with image_result
    ...
)
```

Monadic await

`await` expression is equivalent to:

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

    }
    return await_resume(tmp);
}
```

Answers? Questions! Questions? Answers!

Kudos:

Friends at KDE, Dr Saša Malkov, basysKom

Worth reading and watching:

- Iterators Must Go, Andrei Alexandrescu
- Value Semantics and Range Algorithms, Chandler Carruth
- Systematic Error Handling in C++, Andrei Alexandrescu
- Await 2.0, Gor Nishanov
- Ranges proposal, Eric Niebler
- Reactive manifesto, Books on Erlang or Scala/Akka