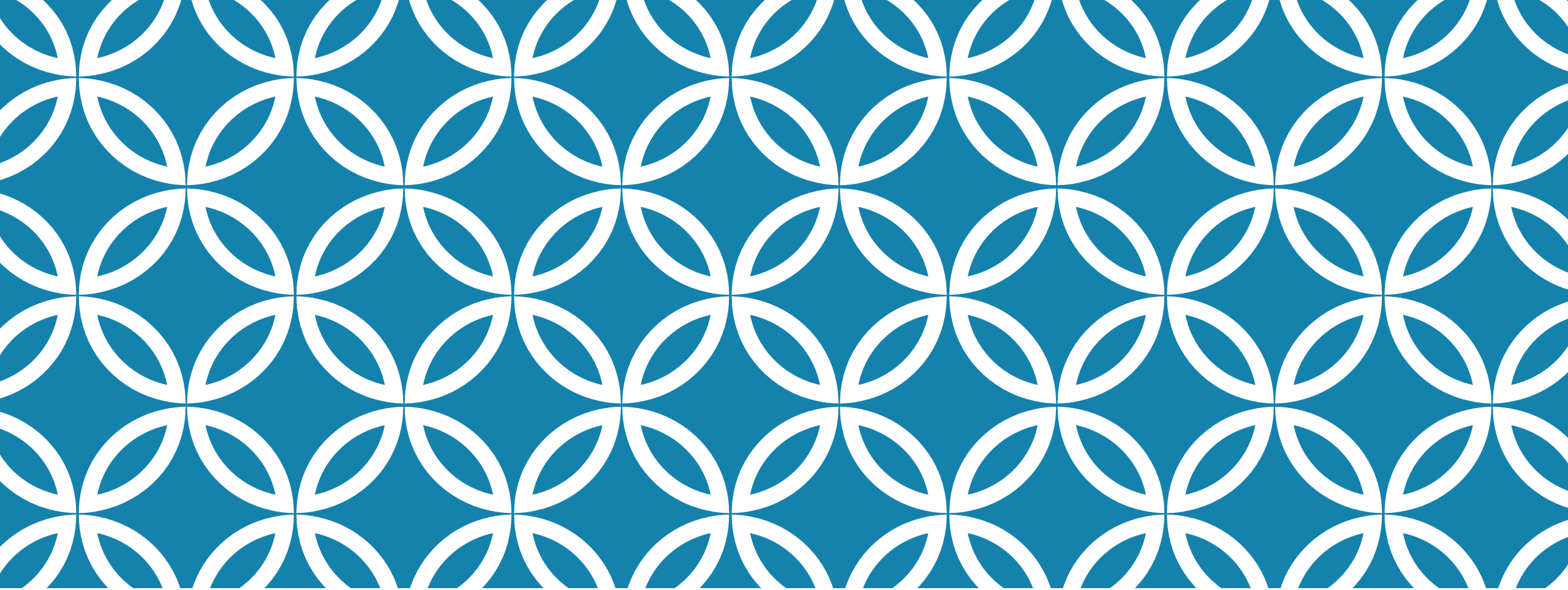


STACKLESS COROUTINES

As a macro-free library

LETS MAKE AN ASYNC TCP ECHO SERVER

```
class session
  : public std::enable_shared_from_this<session>
{
  tcp::socket socket_;
  enum { max_length = 1024 };
  char data_[max_length];

public:
  session(tcp::socket socket)
    : socket_(std::move(socket))
  {
  }

  void start()
  {
    do_read();
```

LETS MAKE AN ASYNC TCP ECHO SERVER

```
private:
  void do_read()
  {
    auto self(shared_from_this());
    socket_.async_read_some(boost::asio::buffer(data_, max_length),
        [this, self](boost::system::error_code ec, std::size_t length)
        {
          if (!ec)
          {
            do_write(length);
          }
        });
  }

  void do_write(std::size_t length)
  {
```

LETS MAKE AN ASYNC TCP ECHO SERVER

```
void do_write(std::size_t length)
{
  auto self(shared_from_this());
  boost::asio::async_write(socket_, boost::asio::buffer(data_, length),
      [this, self](boost::system::error_code ec, std::size_t /*length*/)
      {
        if (!ec)
        {
          do_read();
        }
      });
}
```

LETS MAKE AN ASYNC TCP ECHO SERVER

```
class server
{
  tcp::acceptor acceptor_;
  tcp::socket socket_;
public:
  server(boost::asio::io_service& io_service, short port)
    : acceptor_(io_service, tcp::endpoint(tcp::v4(), port)),
      socket_(io_service)
  {
    do_accept();
  }

private:
  void do_accept()
  {
    acceptor_.async_accept(socket_,
        [this](boost::system::error_code ec)
        {
```

LETS MAKE AN ASYNC TCP ECHO SERVER

```
  }

private:
  void do_accept()
  {
    acceptor_.async_accept(socket_,
        [this](boost::system::error_code ec)
        {
          if (!ec)
          {
            std::make_shared<session>(std::move(socket_))->start();
          }

          do_accept();
        });
  }
```

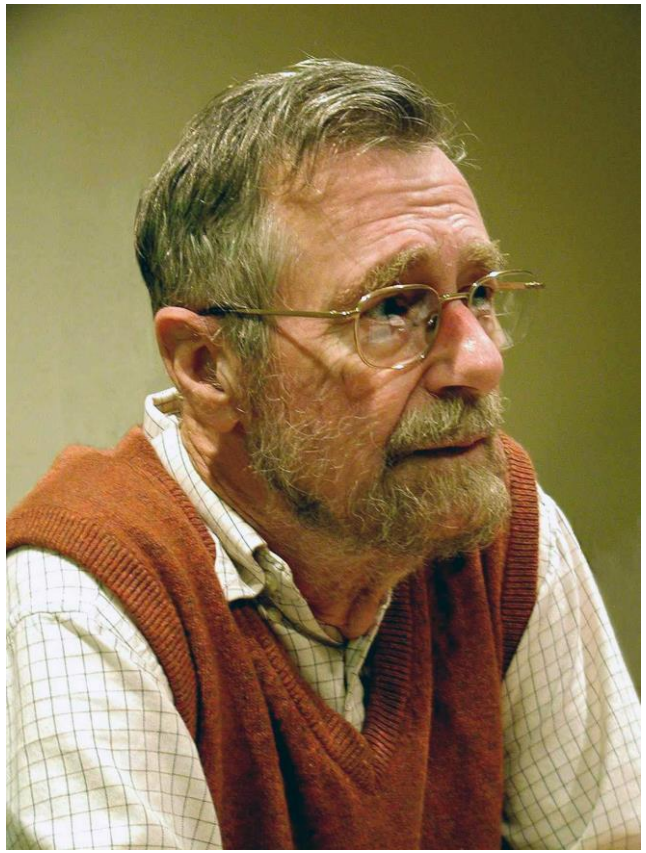
SOME OBSERVATIONS

- Shared pointer makes memory management easier
- Runtime organization, does not reflect source organization
- Error handling can be tricky
- Loops can be tricky

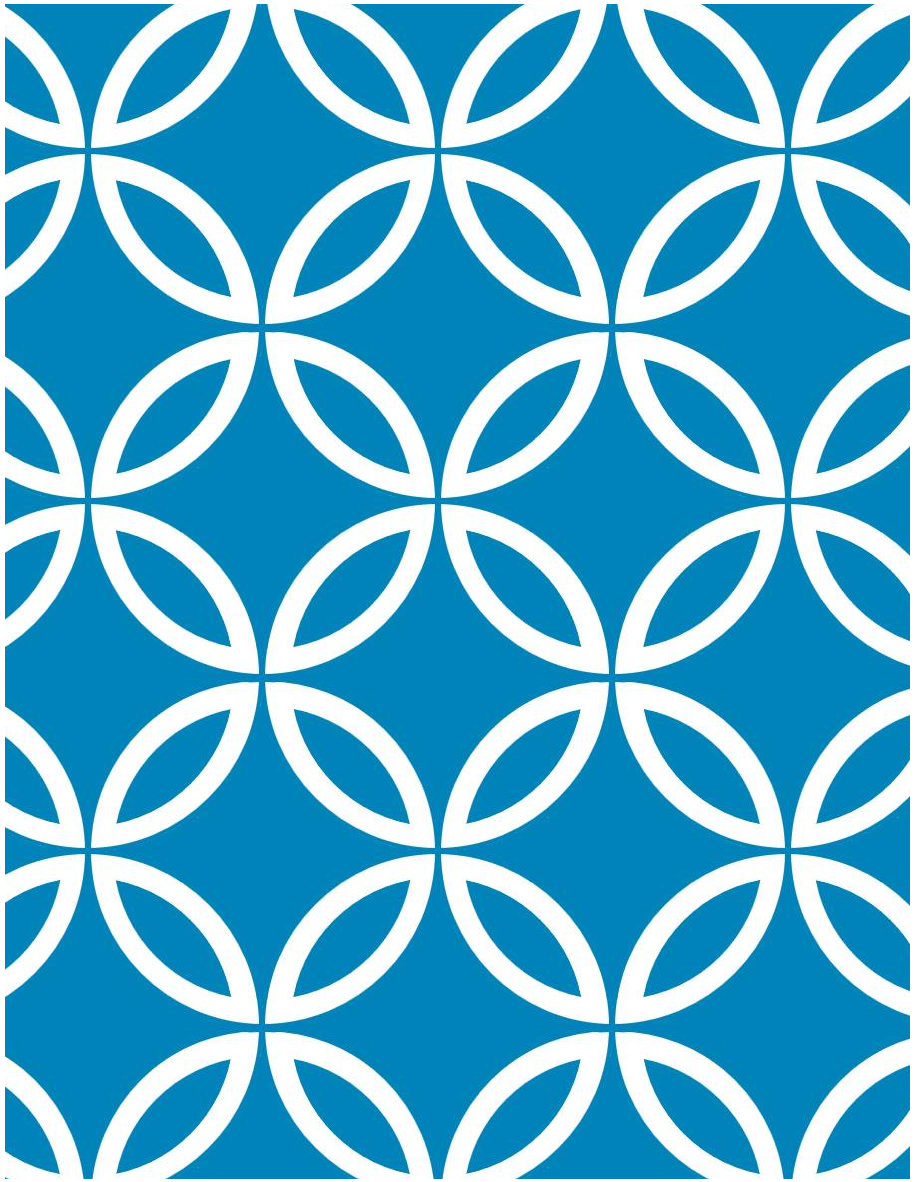
WHAT'S WRONG WITH CALLBACKS

- Hidden structure
- Error handling

HIDDEN STRUCTURE



The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question becuase it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful: in such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, "n" equals the number of persons in the room minus one!



CALLBACKS ARE THE GOTOS OF ASYNCHRONOUS PROGRAMMING

ALTERNATIVES TO CALLBACKS

- Futures/promises
- Coroutines
  - Stackfull
  - Stackless

CALLBACK HELL

```
1   auto result = std::make_shared<double>();
2   fooAsync(input, [=](Output output) {
3     // ...
4     asyncA(output, [=](OutputA outputA) {
5       // ...
6       asyncB(outputA, [=](OutputB outputB) {
7         // ...
8         asyncC(outputB, [=](OutputC outputC) {
9           // ...
10          asyncD(outputC, [=](OutputD outputD) {
11            *result = outputD * M_PI;
12          });
13        });
14      });
15    });
16  });
17  // from https://code.facebook.com/posts/1661982097368498
18
```

FUTURES

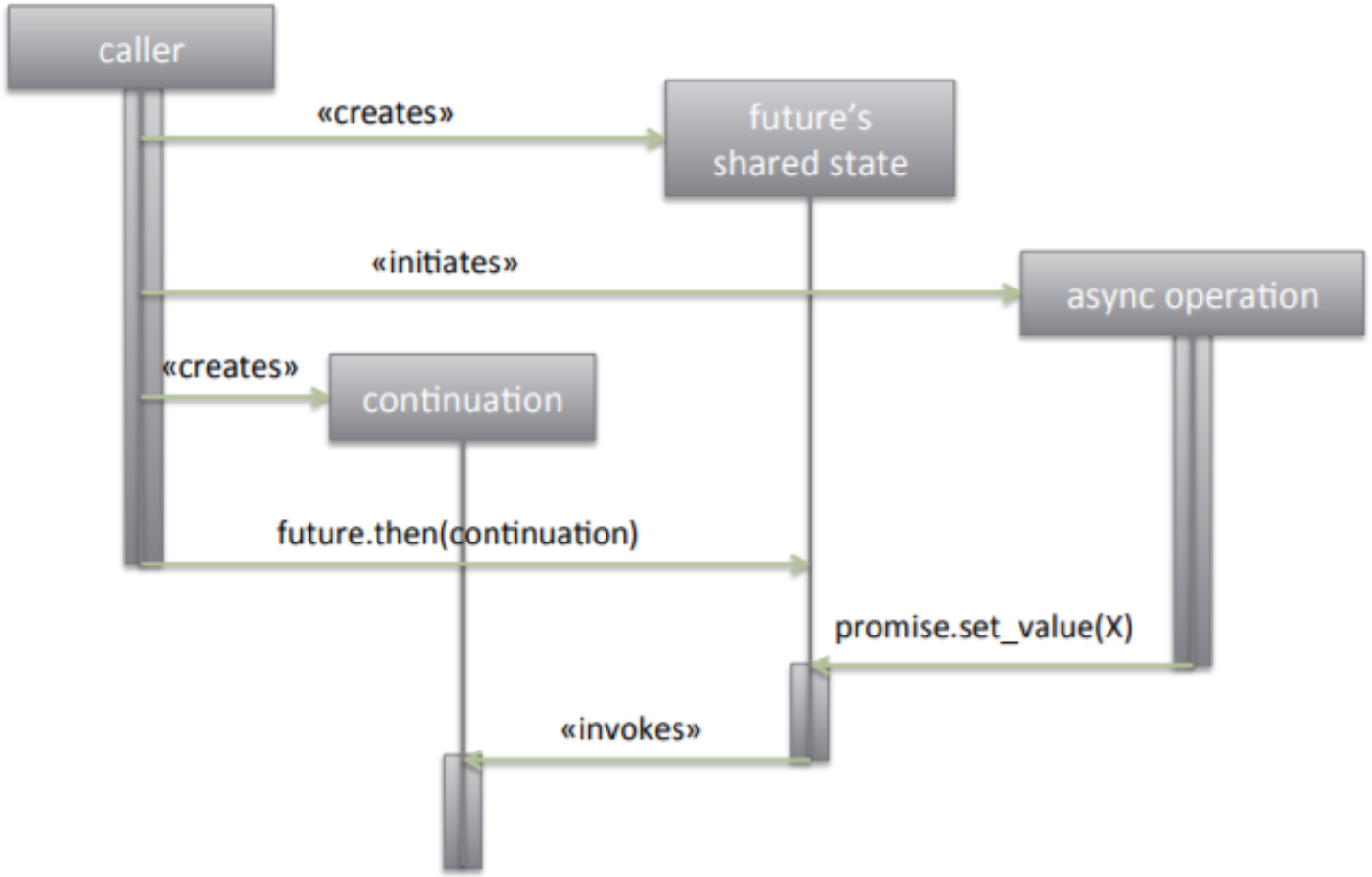
```
1    Future<double> fut =
2      fooFuture(input)
3      .then(futureA)
4      .then(futureB)
5      .then(futureC)
6      .then(d)
7      .then([](OutputD outputD) { // lambdas are ok too
8        return outputD * M_PI;
9      });
10   // From https://code.facebook.com/posts/1661982097368498
11
```

ISSUES WITH FUTURES

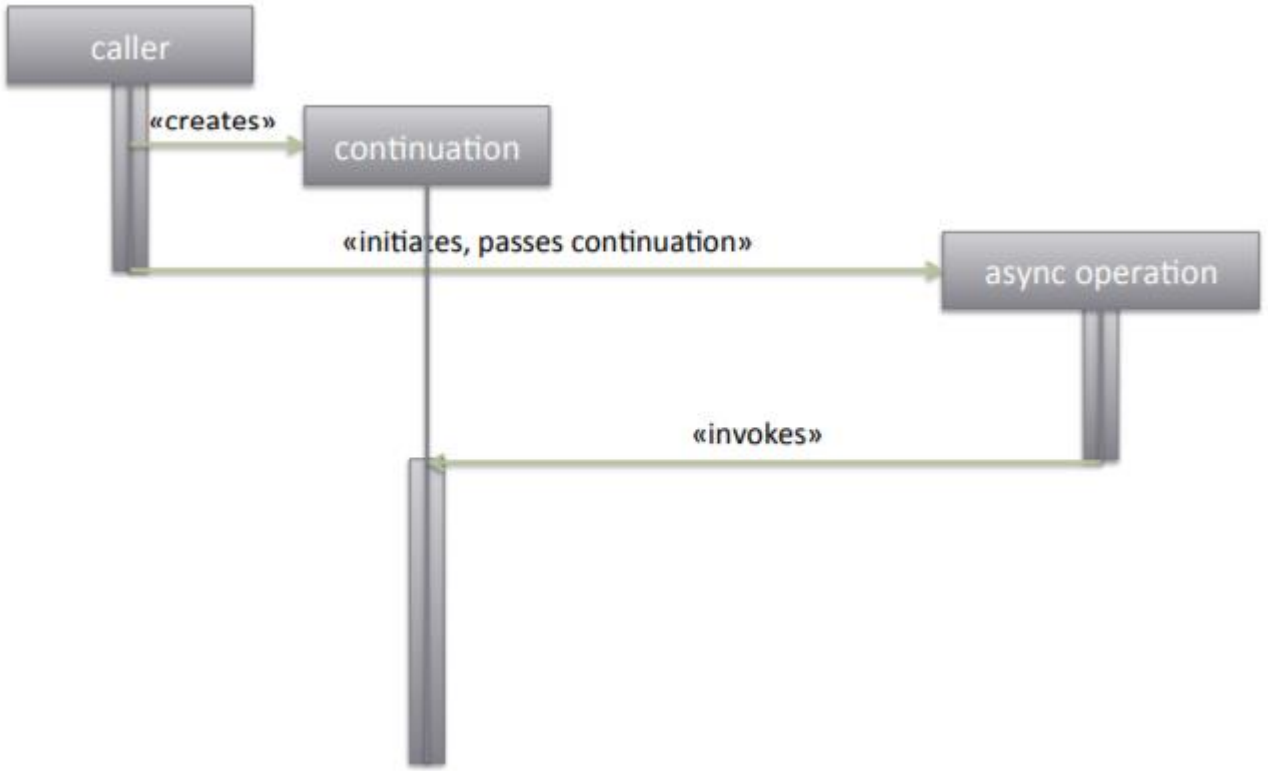
- Performance
- Loops

FUTURES - PERFORMANCE

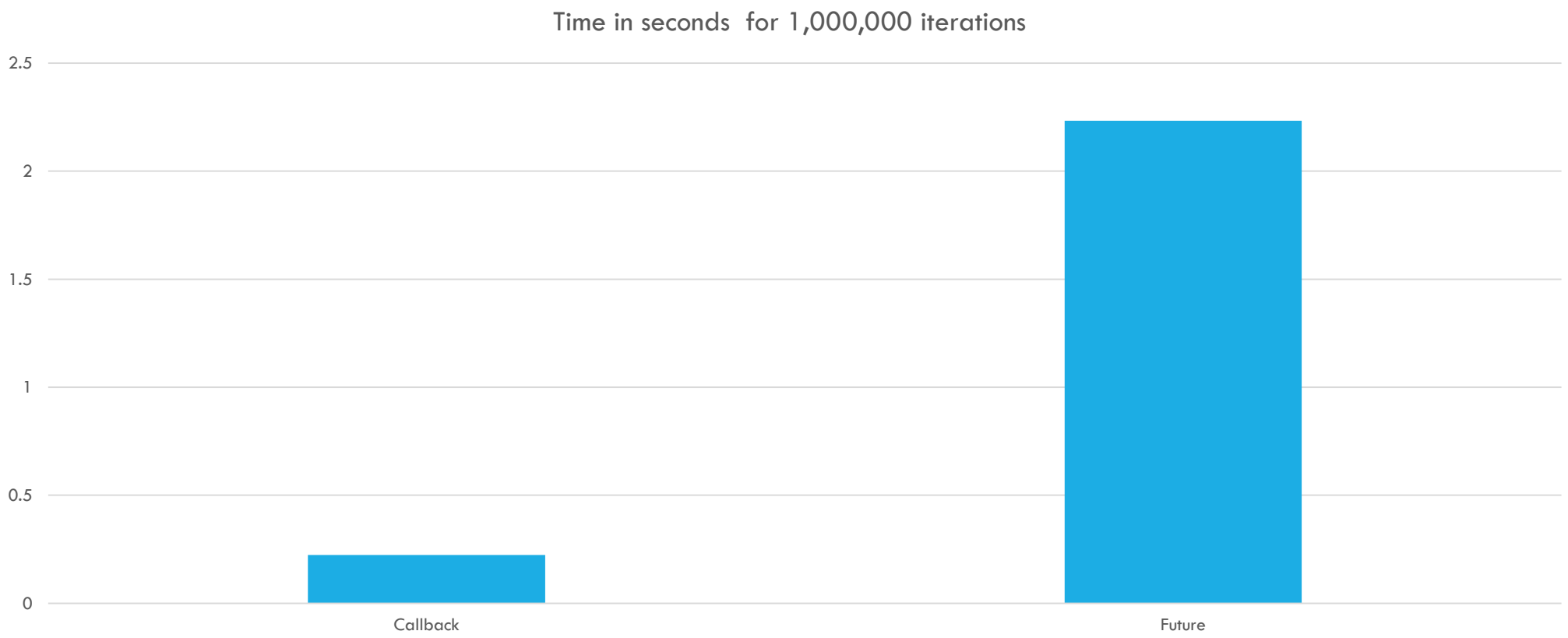
N3747 - A Universal Model for Asynchronous Operations



FUTURES PERFORMANCE



PERF NUMBER FROM CHANNELS TALK



FUTURES LOOPING — ILLUSTRATED BY GOR NISHANOV

Trivial if synchronous

```
int tcp_reader(int total)
{
    char buf[4 * 1024];
    auto conn = Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```


From C++ Coroutines – Gor Nishanov - CppCon 2015

```
future<int> tcp_reader(int64_t total) {
    struct State {
        char buf[4 * 1024];
        int64_t total;
        Tcp::Connection conn;
        explicit State(int64_t total) : total(total) {}
    };
    auto state = make_shared<State>(total);
    return Tcp::Connect("127.0.0.1", 1337).then(
        [state](future<Tcp::Connection> conn) {
            state->conn = std::move(conn.get());
            return do_while([state]()->future<bool> {
                if (state->total <= 0) return make_ready_future(false);
                return state->conn.read(state->buf, sizeof(state->buf)).then(
                    [state](future<int> nBytesFut) {
                        auto nBytes = nBytesFut.get()
                        if (nBytes == 0) return make_ready_future(false);
                        state->total -= nBytes;
                        return make_ready_future(true);
                    });
            });
    });
}
```

N4399 Working Draft, Technical Specification for C++ Extensions for Concurrency

.then

```
future<void> do_while(function<future<bool>()> body) {
    return body().then([=](future<bool> notDone) {
        return notDone.get() ? do_while(body) : make_ready_future(); });
}
```

COROUTINES TO THE RESCUE

Document Number:	P0057R5
Date:	2016-07-10
Revises:	P0057R4
Audience:	CWG / LWG
Authors:	Gor Nishanov <gorn@microsoft.com>
	Jens Maurer <Jens.Maurer@gmx.net>
	Richard Smith <richard@metafoo.co.uk>
	Daveed Vandevoorde <daveed@edg.com>

Wording for Coroutines

WHAT ARE COROUTINES

"Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes."

- Wikipedia "Coroutine"

Trivial

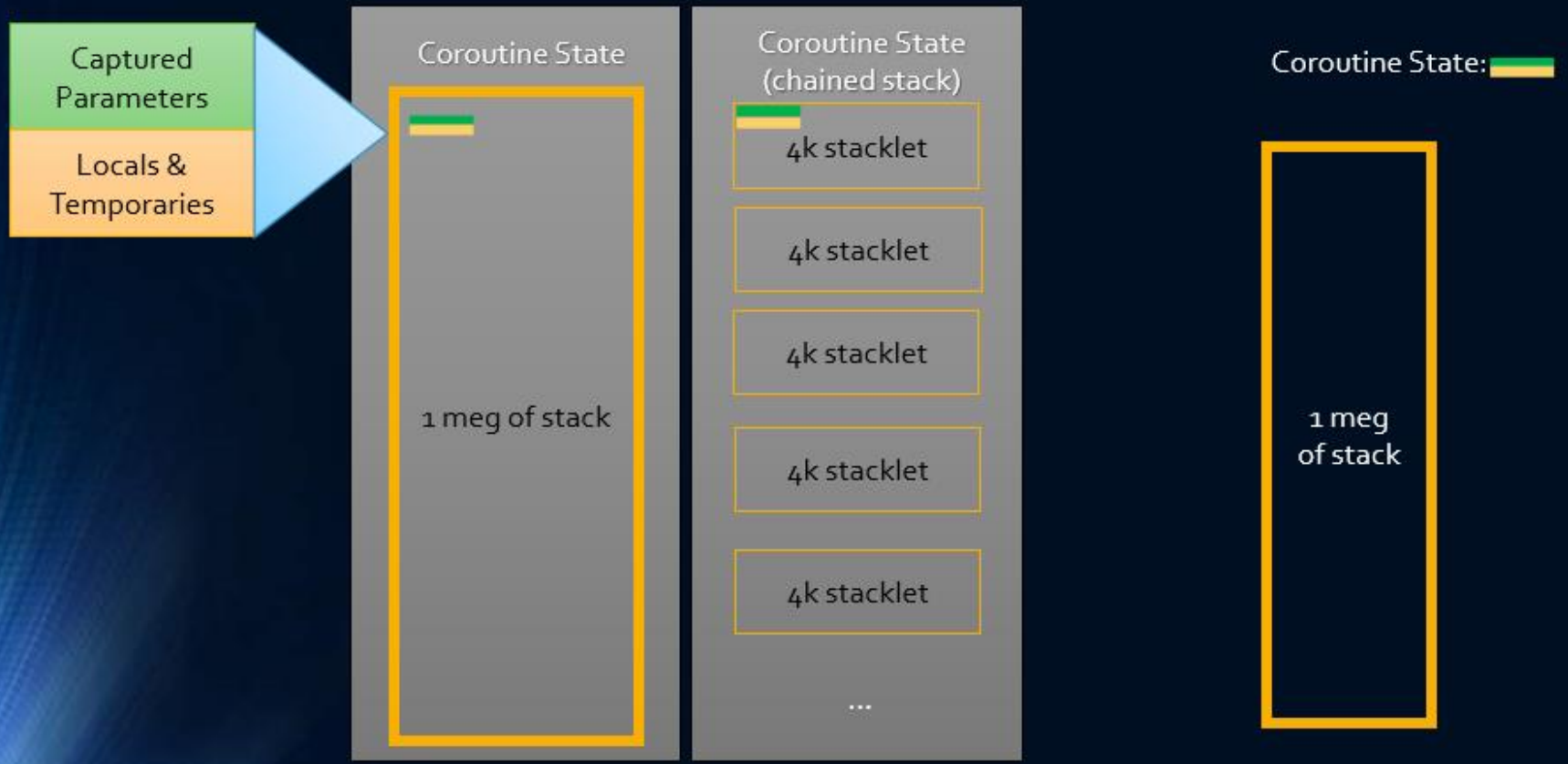
```
auto tcp_reader(int total) -> future<int>
{
    char buf[4 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

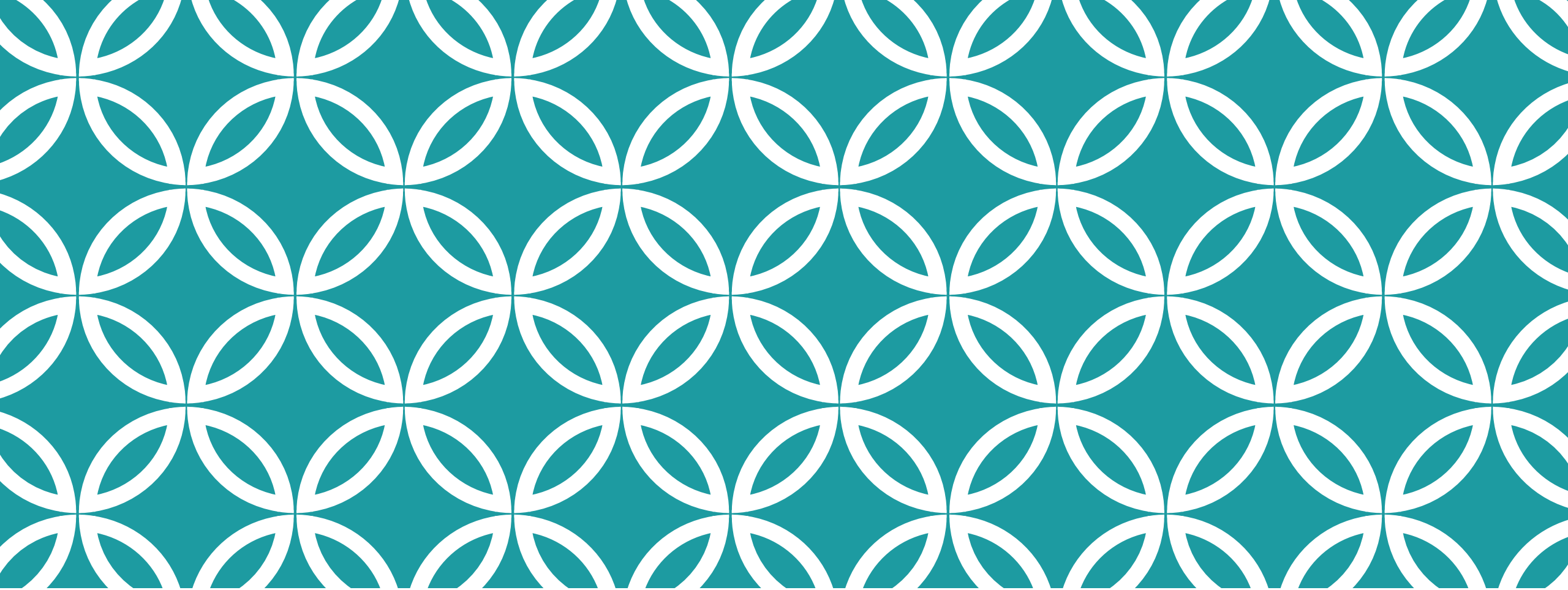
C++17 DOES NOT HAVE COROUTINES ☹

TYPES OF COROUTINES

	Stackless	Stackfull
Language	Coroutine TS (PDTS) Resumable expressions (n4453)	
Library with assembler		Boost Coroutine
Library with macros	CO2 Boost ASIO	
Library with standard C++ and no macros	(this)	

Stackful vs. Stackless





INTRODUCING STACKLESS COROUTINE LIBRARY

WHAT C++ FEATURE IS THIS?

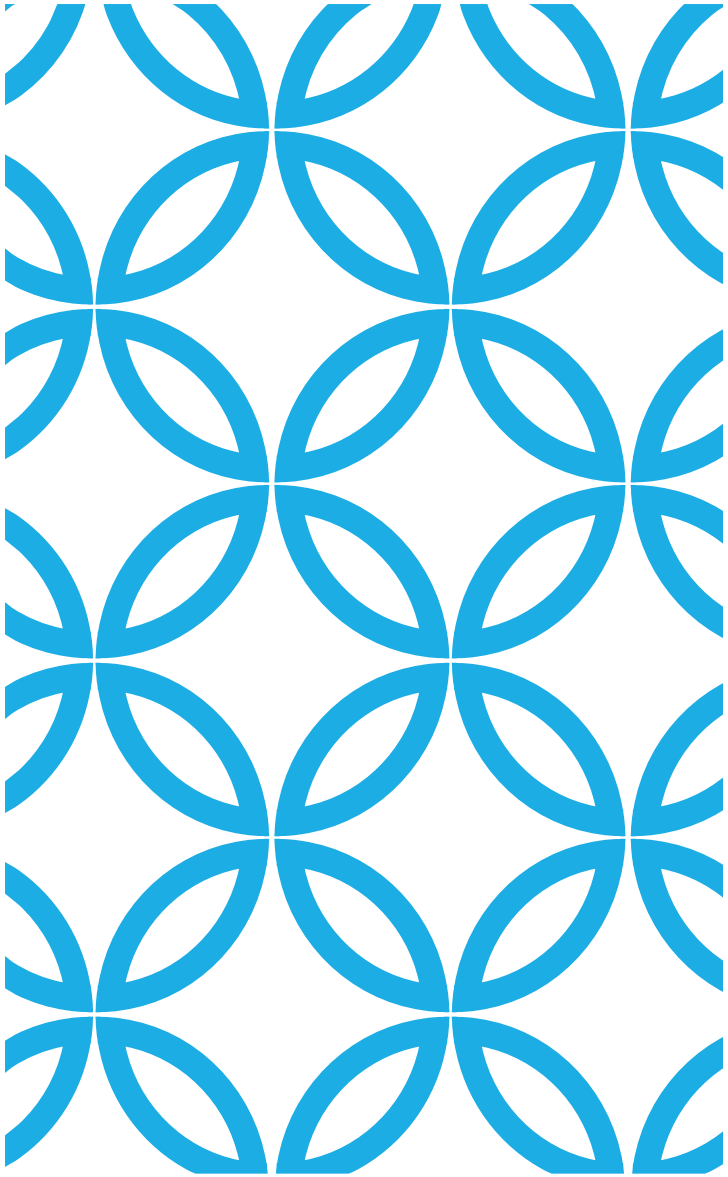
DUH!

WHAT C++ FEATURE IS THIS?

DUH!

STACKLESS COROUTINE LIBRARY

- C++14
- No macros
- Header-only
- Works with and without exceptions
- No dependencies
- Drop in replacement for callbacks
- Supports generators



STARRING TUPLE AND GENERIC LAMBDA

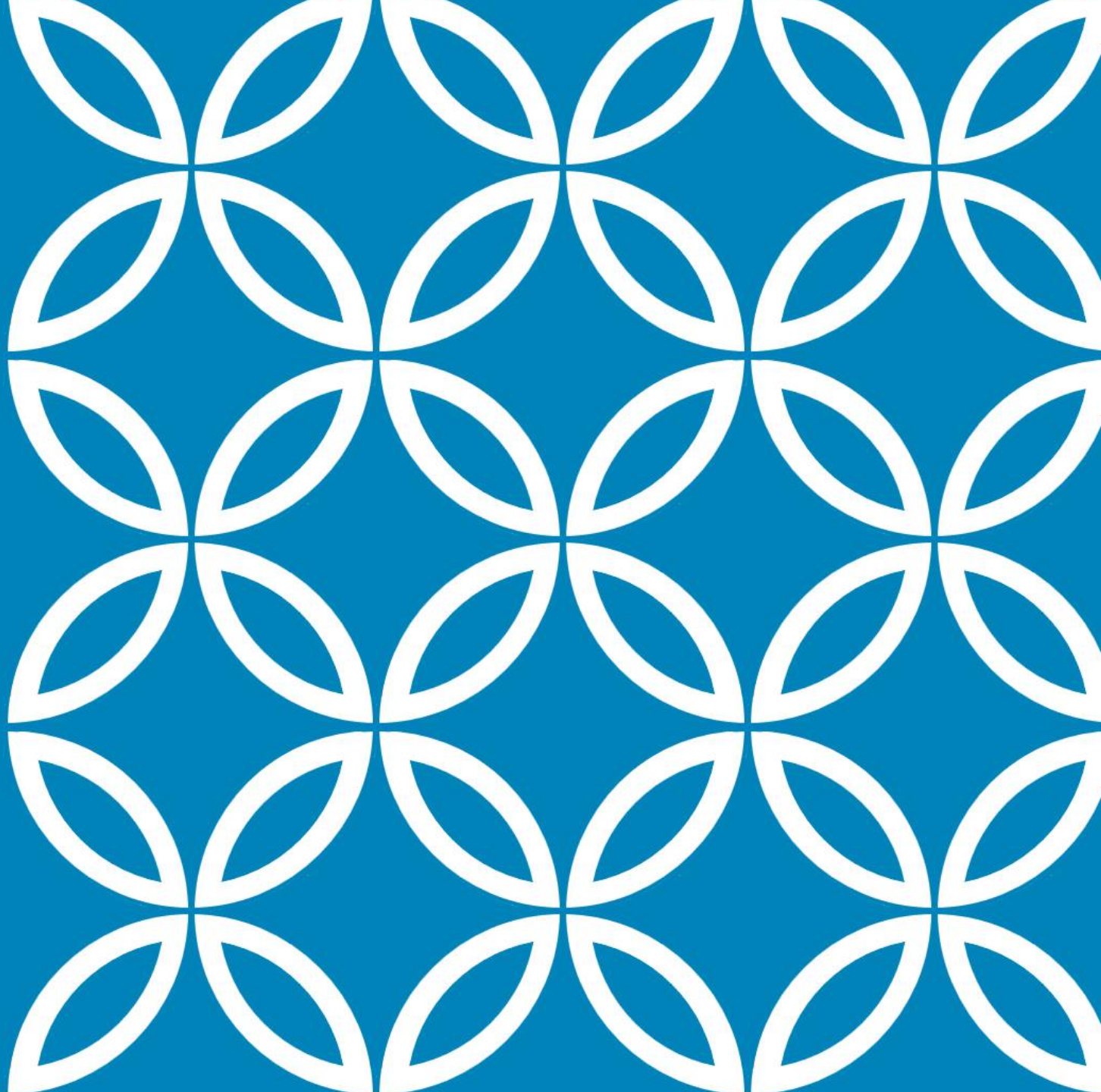
CENTRAL IDEA – A TUPLE OF…

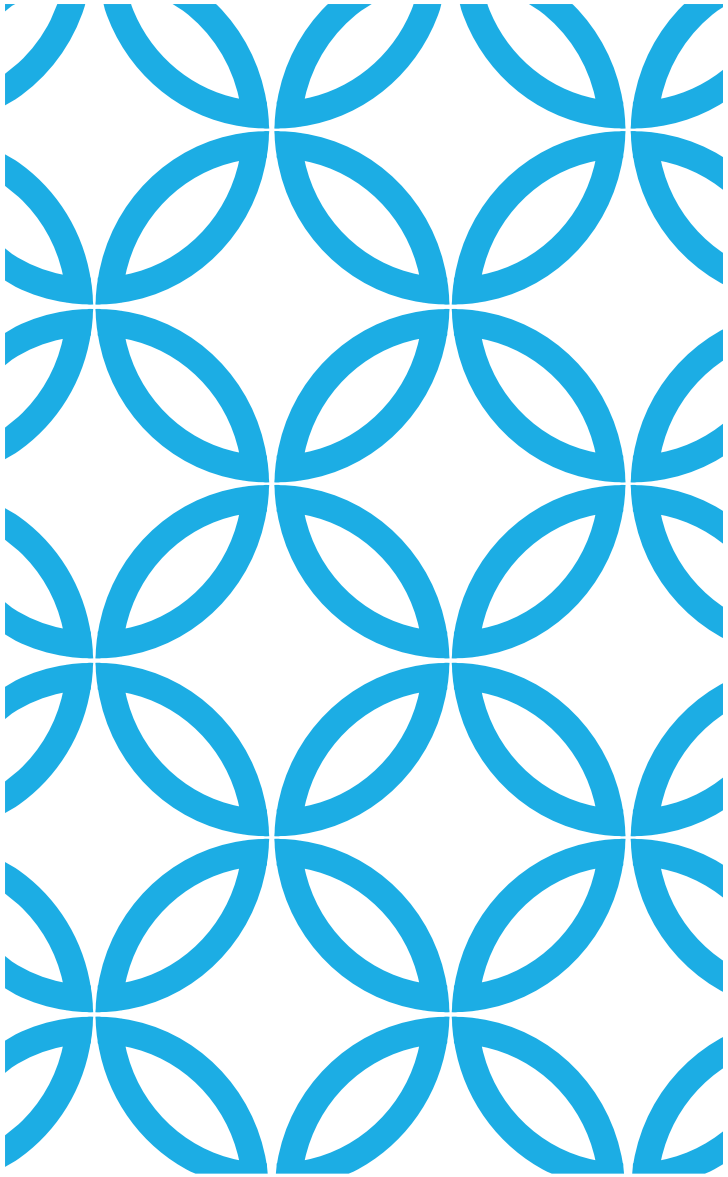
1. Generic Lambda
2. If_t
    1. Generic Lambda
    2. Tuple
        1. Generic Lambda
        2. Generic Lambda
    3. Tuple
        1. Generic Lambda
3. While_true_t
    1. Generic Lambda
    2. Generic Lambda

COMBINED WITH

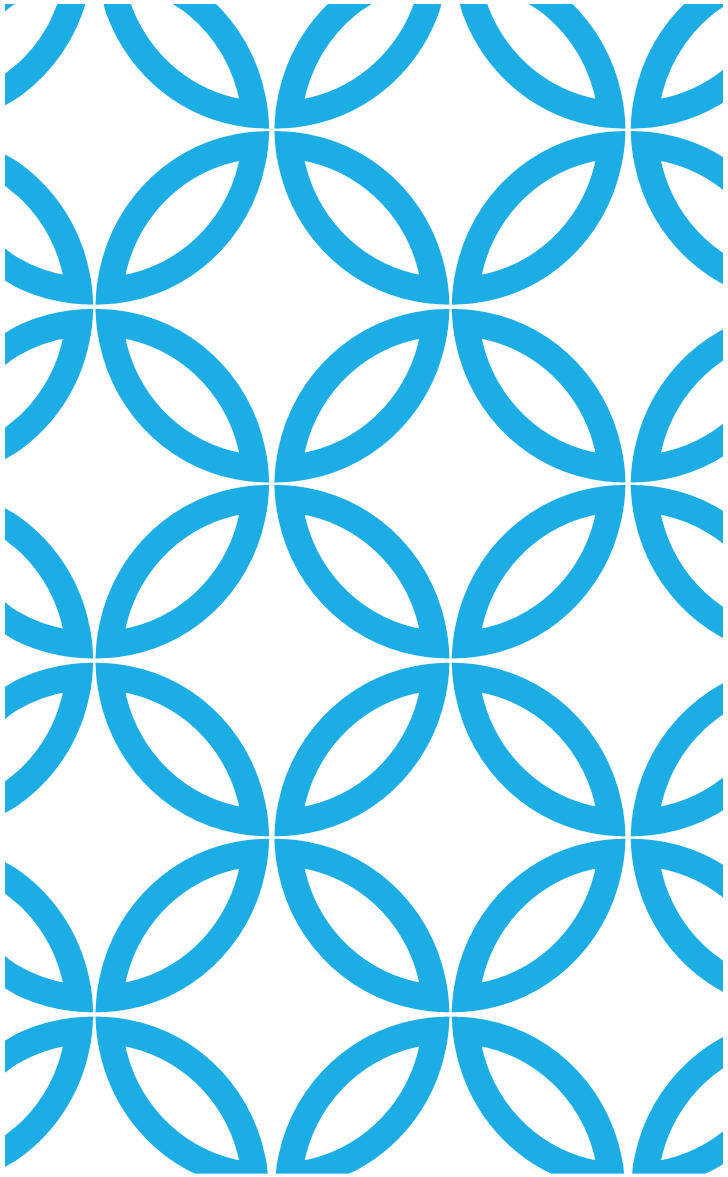
- Variables struct
  - Variable1
  - Variable2

ASIDE: 3 C++ DECISIONS THAT EXPLAIN EVERYTHING

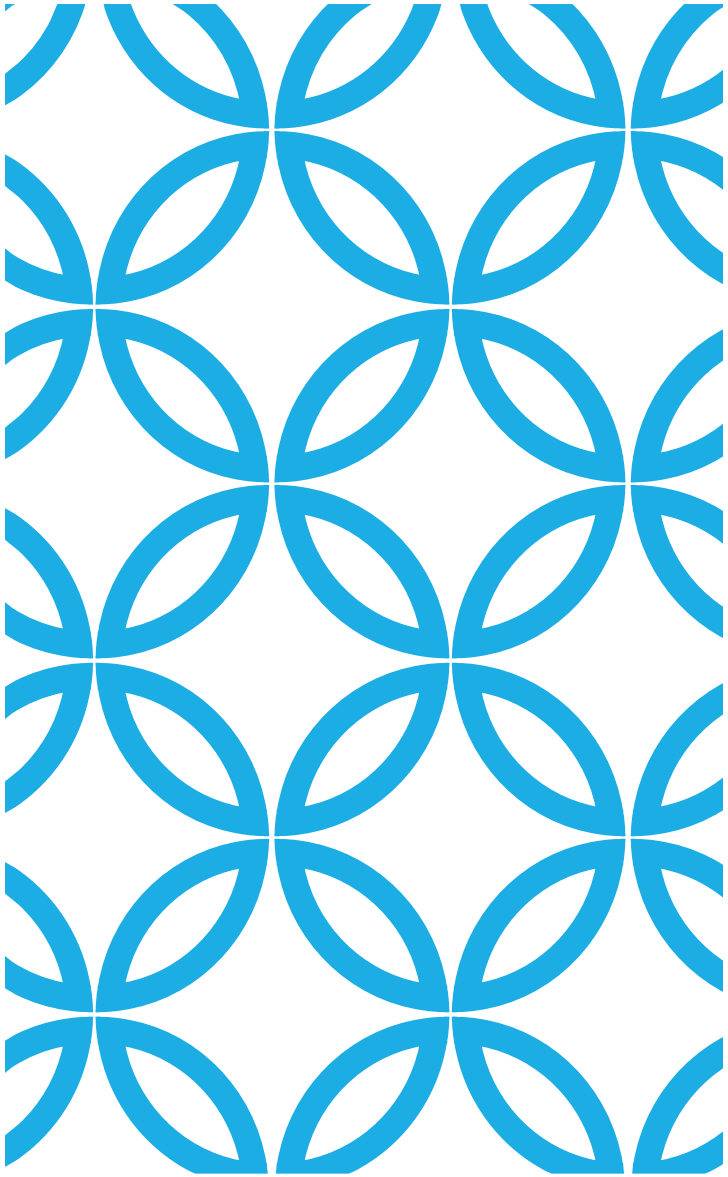




BASED ON C



STRING AS A LIBRARY TYPE



TUPLE AS A LIBRARY TYPE

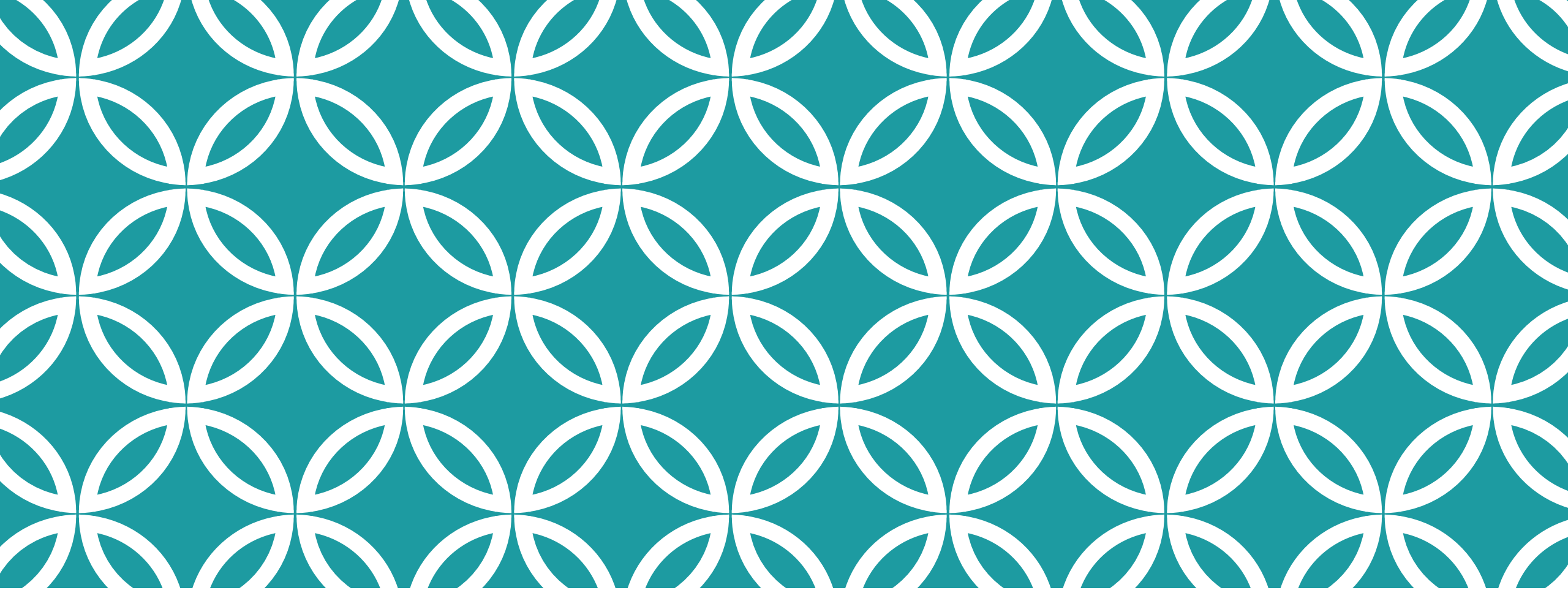
ALMOST EVERYTHING IN C++ IS BECAUSE

Required for C compatibility

Makes it easier to use/efficiently implement string

Makes it easy to use/efficiently implement tuple

(Multithreading)



EXAMPLES

ASYNC TCP ECHO SERVER - SESSION

```
void session(tcp::socket socket) {

  constexpr int max_length = 1024;
  struct session_variables {
    tcp::socket socket;
    char data[max_length];
    explicit session_variables(tcp::socket s) : socket{std::move(s)} {}
  };

  static auto block =
      stackless_coroutine::make_block(stackless_coroutine::make_while_true(
          [](auto &context, auto &variables) {
            variables.socket.async_read_some(
                boost::asio::buffer(variables.data, max_length), context);
            return context.do_async();
          },
```

```
  static auto block =
      stackless_coroutine::make_block(stackless_coroutine::make_while_true(
          [](auto &context, auto &variables) {
            variables.socket.async_read_some(
                boost::asio::buffer(variables.data, max_length), context);
            return context.do_async();
          },
          [](auto &context, auto &variables, auto &ec, auto length) {
            if (ec)
              return context.do_async_break();
            boost::asio::async_write(
                variables.socket, boost::asio::buffer(variables.data, length),
                context);
            return context.do_async();
          },
          [](auto &context, auto &variables, auto &ec, auto) {
            if (ec)
              return context.do_break();
            return context.do_next();
          }));
  auto co = stackless_coroutine::make_coroutine<session_variables>(
      block, [](auto &&...) {}, std::move(socket));
  co();
}
```

ASYNC TCP ECHO SERVER - SERVER

```
void server(boost::asio::io_service &io_service, short port) {
  struct server_variables {
    tcp::acceptor acceptor;
    tcp::socket socket;
    server_variables(boost::asio::io_service &io_service, short port)
        : acceptor(io_service, tcp::endpoint(tcp::v4(), port)),
          socket(io_service) {}
  };

  static auto block =
      stackless_coroutine::make_block(stackless_coroutine::make_while_true(
          [](auto &context, auto &variables) {
            variables.acceptor.async_accept(variables.socket, context);
            return context.do_async();
          },
          [](auto &context, auto &variables, auto &ec) {
            if (!ec) {
```

ASYNC TCP ECHO SERVER- SERVER

```
    static auto block =
        stackless_coroutine::make_block(stackless_coroutine::make_while_true(
            [](auto &context, auto &variables) {
              variables.acceptor.async_accept(variables.socket, context);
              return context.do_async();
            },
            [](auto &context, auto &variables, auto &ec) {
              if (!ec) {
                session(std::move(variables.socket));
              }
            }));
    stackless_coroutine::make_coroutine<server_variables>(
        block, [](auto &&...) {}, io_service, port)();
}
```

ASYNC TCP ECHO SERVER - MAIN

```
1   int main(int argc, char *argv[]) {
2     try {
3       if (argc != 2) {
4         std::cerr << "Usage: async_tcp_echo_server <port>\n";
5         return 1;
6       }
7
8       boost::asio::io_service io_service;
9
10      server(io_service, std::atoi(argv[1]));
11
12      io_service.run();
13    } catch (std::exception &e) {
14      std::cerr << "Exception: " << e.what() << "\n";
15    }
```

RECURSIVE DIRECTORY ITERATOR

```
1   generator<std::string> make_recursive_directory_generator(
2       const filesystem::path &p) {
3
4     struct variables_t {
5       filesystem::directory_iterator begin;
6       filesystem::directory_iterator end;
7       generator<std::string> gen;
8
9       variables_t(const filesystem::path &path)
10          : begin{path} {}
11    };
12
13    auto block = make_block(stackless_coroutine::make_while(
14        [](auto &variables) { return variables.begin != variables.end; },
15        [](auto &context, auto &variables) {
16          return context.do_async_yield(variables.begin->path().string());
```

```
auto block = make_block(stackless_coroutine::make_while(
    [](auto &variables) { return variables.begin != variables.end; },
    [](auto &context, auto &variables) {
      return context.do_async_yield(variables.begin->path().string());
    },
    stackless_coroutine::make_if(
        [](auto &variables) {
          return filesystem::is_directory(variables.begin->path());
        },
        std::make_tuple(
            [](auto &context, auto &v) {
              v.gen = make_recursive_directory_generator(v.begin->path());
            },
            stackless_coroutine::make_while(
                [](auto &variables) {
                  return variables.gen.begin() != variables.gen.end();
                },
                [](auto &context, auto &variables) {
                  return context.do_async_yield(
                      std::move(*variables.gen.begin()));
                },
                [](auto &context, auto &variables) {
                  ++variables.gen.begin();
                })),
        std::make_tuple()),
    [](auto &context, auto &variables) { ++variables.begin; }));
return make_generator<string, variables_t>(block,p);
```

USING THE GENERATOR

```
1     auto g = make_recursive_directory_generator(".");
2     std::vector<std::string> vgen(g.begin(), g.end());
3
```

ASYNC HTTP CLIENT

```
1   template <class Finished>
2   auto print_url(boost::asio::io_service &io, std::string host, std::string url,
3                  Finished f) {
4
5     struct coroutine_variables_t {
6       boost::asio::ip::tcp::resolver resolver;
7       boost::asio::ip::tcp::socket socket;
8       boost::asio::streambuf request;
9       boost::asio::streambuf response;
10      bool eof = false;
11      explicit coroutine_variables_t(boost::asio::io_service &io)
12          : resolver{io}, socket{io} {}
13    };
14
```

CREATING THE BLOCK

```
15      // Create the block for the coroutine
16      static auto block = stackless_coroutine::make_block(
17
18          [](auto &context, auto &variables, const std::string &host,
19             const std::string &path) {
20
21            // Start an asynchronous resolve to translate the server and service
22            // names into a list of endpoints.
23            boost::asio::ip::tcp::resolver::query query{host, "http"};
24            variables.resolver.async_resolve(query, context);
25
26            return context.do_async();
27          },
28          [](auto &context, auto &variables, auto &ec, auto iterator) {
29            throw_if_error(ec);
30            // Attempt a connection to each endpoint in the list until we
31            // successfully establish a connection.
32            boost::asio::async_connect(variables.socket, iterator, context);
```

ASYNC

```
    },
    [](auto &context, auto &variables, auto ec, auto iterator) {
      throw_if_error(ec);
      // Attempt a connection to each endpoint in the list until we
      // successfully establish a connection.
      boost::asio::async_connect(variables.socket, iterator, context);
      return context.do_async();
    },
    [](auto &context, auto &variables, auto &ec, auto iterator) {
      throw_if_error(ec);
      // The connection was successful. Send the request.

      boost::asio::async_write(variables.socket, variables.request, context);
      return context.do_async();
    },
    [](auto &context, auto &variables, auto &ec, auto n) {
      throw_if_error(ec);
```

LOOPING

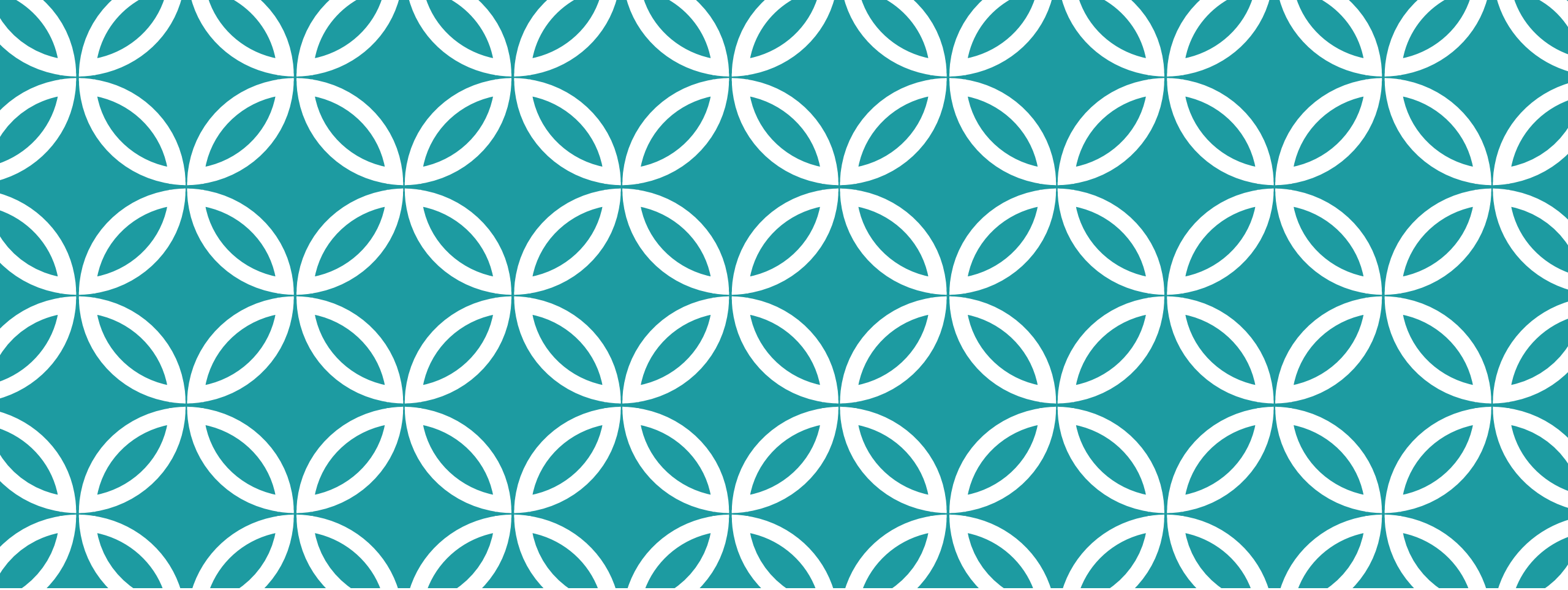
```
        });

        stackless_coroutine::make_while(
            [](auto &variables) { return !variables.eof; },
            [](auto &context, auto &variables) {
              boost::asio::async_read(variables.socket, variables.response,
                                      boost::asio::transfer_at_least(1), context);
              return context.do_async();          },
            [](auto &context, auto &variables, auto &ec, auto n) {
              if (ec == boost::asio::error::eof) {
                variables.eof = true;
              } else {
                throw_if_error(ec);
                std::cout << &variables.response;
              }
            }) // make_while
    );          // make block
```

CREATING THE COROUTINE

```
                                    boost::asio::transfer_at_least(1), context);
            return context.do_async();          },
          [](auto &context, auto &variables, auto &ec, auto n) {
            if (ec == boost::asio::error::eof) {
              variables.eof = true;
            } else {
              throw_if_error(ec);
              std::cout << &variables.response;
            }
          }) // make_while
  );          // make_block
  auto co = stackless_coroutine::make_coroutine<coroutine_variables_t>(
      block, std::move(f), io);

  co(host, url);
}
```

MAIN

```
1    boost::asio::io_service io;
2        print_url(
3            io, argv[1], argv[2],
4
5            [&](auto &variables, std::exception_ptr e, auto op) {
6              if (e) {
7                std::rethrow_exception(e);
8              }
9            });
10
11       io.run();
12     } catch (std::exception &e) {
13       std::cout << "Exception: " << e.what() << "\n";
14     }
15
```



IMPLEMENTING COROUTINES

VARIABLES

```
1   template <class Variables, class Tuple, class Callback>
2   struct variables_t : Variables {
3       template <class... A>
4           variables_t(const Tuple* tup, Callback callback, A&&... a) :
5              Variables{ std::forward<A>(a)... },
6              stackless_coroutine_tuple{ tup },
7              stackless_coroutine_callback_function{ std::move(callback) } {}
8
9       const Tuple* stackless_coroutine_tuple;
10      Callback stackless_coroutine_callback_function;
11  };
12
```

CENTRAL IDEA – A TUPLE OF…

1. Generic Lambda
2. If_t
    1. Generic Lambda
    2. Tuple
        1. Generic Lambda
        2. Generic Lambda
    3. Tuple
        1. Generic Lambda
3. While_true_t
    1. Generic Lambda
    2. Generic Lambda

EVERY GENERIC LAMBDA IS UNIQUELY SPECIFIED BY AN INTEGER SEQUENCE – FOR EXAMPLE 2,2,1

1. Generic Lambda
2. If_t
    1. Generic Lambda
    2. Tuple
        1. Generic Lambda
        2. Generic Lambda
    3. Tuple
        1. Generic Lambda
3. While_true_t
    1. Generic Lambda
    2. Generic Lambda

THE GENERIC LAMBDA

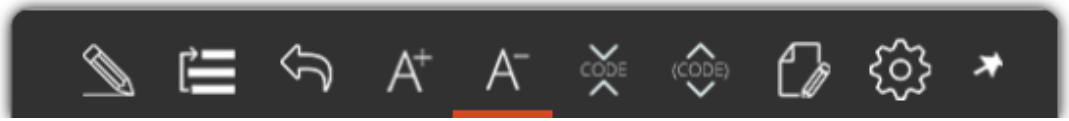
```
1  [](auto &context, auto &variables) {
2
3  }
4
```

CONTEXT

```
template<class Variables> struct dummy_coroutine_context {
    operation do_return();
    operation do_break();
    operation do_continue();
    operation do_next();

    async_result do_async();
    template <class Value> async_result do_async_yield(Value v);
    template <class... T> operator()(T &&... t) { }

    async_result do_async_return();
    async_result do_async_break();
    async_result do_async_continue();

    template <class V> static dummy_coroutine_context get_context(V *v);
};
```

OPERATION AND ASYNC_RESULT

```
1   enum class operation {
2       _suspend,
3       _next,
4       _return,
5       _break,
6       _continue,
7       _done,
8       _exception,
9   };
10
11  struct async_result {
12      operation op = operation::_suspend;
13  };
14
```



THE GENERIC LAMBDA

```
[](auto& context, auto& variables){

}

[](auto& context, auto& variables){
  return context.do_next();
}

[](auto& context, auto& variables){
  return context.do_async();
}
```

PROCESSING A LAMBDA IN A TUPLE

```
template<int N, class FunctionLevelTuple, class OuterSequence,
class Tuple, class Variables, class... Args>
static operation process(type2type<operation>,
FunctionLevelTuple& function_level_tuple,
  OuterSequence outer_sequence, const Tuple& t, Variables& vars, Args&&... args) {

    operation_coroutine_context<Variables> oc{ &vars };

    auto op = get_function<N>(t)( oc, vars, std::forward<Args>(args)...);

    return process_next<N>(op, function_level_tuple, outer_sequence, t, vars);
}
```

PROCESSING NEXT LAMBDA

```
template<int N, class FunctionLevelTuple,
class OuterSequence, class Tuple, class Variables>
static operation process_next(operation op, FunctionLevelTuple& function_level_tuple,
OuterSequence outer_sequence, const Tuple& t, Variables& vars) {
  if (op == operation::_next) {
    return if_else_<N < get_size_v<Tuple> - 1>(
      [&](auto& t) ->operation {
        auto next_return = get_return(vars, get_function<N + 1>(t));
        return process<N + 1>(next_return, function_level_tuple, outer_sequence, t, vars);
      },
      [](auto& t) {return operation::_done; },
      t);
  }
  else {return op;}
}
```

NOW WE CAN PROCESS THIS BLOCK

```
make_block(
  [](auto& context, auto& variables){

  },
  [](auto& context, auto& variables){
     return context.do_next();
  },
  [](auto& context, auto& variables){

  }
);
```

JUMPING TO A LAMBDA IN A TUPLE

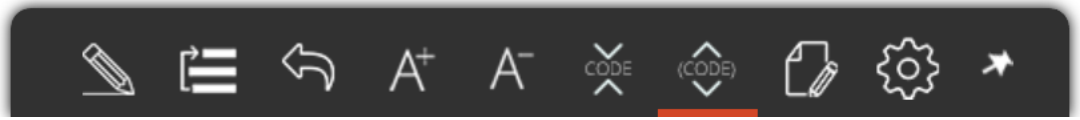
```
template<int N, int Next, int... Rest, class FunctionLevelTuple,
class OuterSequence, class Tuple, class Variables, class... Args>
static operation process_block(integer_sequence<int, N, Next, Rest...>,
 FunctionLevelTuple& function_level_tuple,
 OuterSequence outer_sequence, const Tuple& t,
 Variables& vars, Args&&... args) {

  operation op;
  op = process_block(std::integer_sequence<int, Next, Rest...>{},
    function_level_tuple, append_sequence<N>(outer_sequence),
    get_function<N>(t), vars, std::forward<Args>(args)...);

  if (op == operation::_done) {
    return if_else_ < (N < get_size_v<Tuple> - 1)>([&](auto& t) {
    return process_block(std::integer_sequence<int, N + 1>{},
```

JUMPING TO A LAMBDA IN A TUPLE

```
  operation op;
  op = process_block(std::integer_sequence<int, Next, Rest...>{},
    function_level_tuple, append_sequence<N>(outer_sequence),
    get_function<N>(t), vars, std::forward<Args>(args)...);

  if (op == operation::_done) {
    return if_else_ < (N < get_size_v<Tuple> - 1)>([&](auto& t) {
    return process_block(std::integer_sequence<int, N + 1>{},
        function_level_tuple, outer_sequence, t, vars);
      },
      [] {return operation::_done; }, t);
   }
   return op;
}
```

JUMPING TO A LAMBDA IN A TUPLE

```
1    template<int N, class FunctionLevelTuple, class OuterSequence,
2    class Tuple, class Variables, class... Args>
3    static operation process_block(std::integer_sequence<int, N>,
4    FunctionLevelTuple& function_level_tuple, OuterSequence outer_sequence,
5    const Tuple& t, Variables& vars, Args&&... args) {
6
7      auto return_type = get_return(vars, get_function<N>(t), std::forward<Args>(args)...);
8      return process<N>(return_type, function_level_tuple, outer_sequence,
9        t, vars, std::forward<Args>(args)...);
10   }
11
```



WE CAN JUMP DIRECTLY TO LINE 6

```
1    make_block(
2     [](auto& context, auto& variables){},
3     make_if([](auto& variables}{return true;},
4       std::make_tuple(
5         [](auto& context, auto& variables){},
6         [](auto& context, auto& variables){};
7       ),
8       std::make_tuple(
9         [](auto& context, auto& variables){}
10      )
11    )
12   );
13
14   process_block(std::integer_sequence<int,1,1>{},...);
15
```

ERROR HANDLING

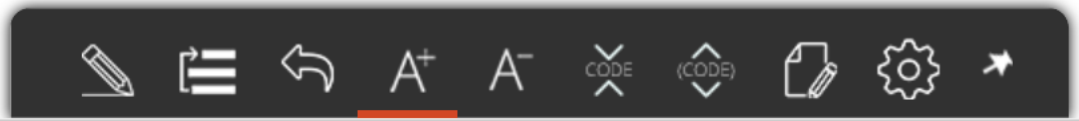
```
1   template<int N, int... Rest, class FunctionLevelTuple,
2   class Variables, class... Args>
3   static void process_outermost_block(std::integer_sequence<int, N, Rest...> seq,...) {
4
5     operation op;
6     exception_ptr eptr{ nullptr };
7     try {
8       op = process_block(seq, function_level_tuple, std::integer_sequence<int>{},
9         function_level_tuple, vars, std::forward<Args>(args)...);
10    }
11    catch (...) {
12      eptr = std::current_exception();
13      op = operation::_exception;
14    }
15
16    if (op == operation::_done || op == operation::_return || op == operation::_exception) {
17      vars.stackless_coroutine_callback_function(vars, eptr, op);
18    }
19  }
```

ASYNC OPERATIONS - CONTEXT

```
template<class Sequence, class Variables>
struct async_coroutine_context {
  Variables* pv;
  template <class... A> void operator()(A &&... a) {
    process_outermost_block(Sequence{},
      *pv->stackless_coroutine_tuple, *pv, std::forward<A>(a)...);
  }

   async_result do_async() { return async_result{ operation::_suspend }; }

  // Other stuff
};
```

PROCESSING ASYNC LAMBDAS

```
template<int N, class FunctionLevelTuple, class OuterSequence,
class Tuple, class Variables, class... Args>
static operation process(type2type<async_result>, FunctionLevelTuple& function_level_tuple,
  OuterSequence outer_sequence, const Tuple& t, Variables& vars, Args&&... args) {

  async_coroutine_context<
    decltype(append_sequence<N + 1>(outer_sequence)), Variables
  > ac{ &vars };
  async_result op = get_function<N>(t)(ac,vars, std::forward<Args>(args)...);
  return op.op;
}
```



NOW THIS WORKS

```
make_block(
    // ...
    [](auto &context, auto &variables, auto ec, auto iterator) {
        throw_if_error(ec);
        boost::asio::async_connect(variables.socket, iterator, context);
        return context.do_async();
      },
      [](auto &context, auto &variables, auto &ec, auto iterator) {
        throw_if_error(ec);
        boost::asio::async_write(variables.socket, variables.request, context);
        return context.do_async();
      }
      //...
);
```

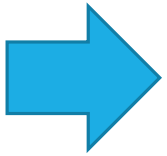
LOOPS

```
template<class Tuple>
struct while_t {
    Tuple t;
};
template<class T>
struct is_while_t { enum { value = 0 }; };
template<class T>
struct is_while_t<while_t<T>> {
  enum {value = 1};
};
```

RECALL OUR PROCESS BLOCK FUNCTION

```
template<int N, int Next, int... Rest, class FunctionLevelTuple,
class OuterSequence, class Tuple, class Variables, class... Args>
static operation process_block(integer_sequence<int, N, Next, Rest...>,
 FunctionLevelTuple& function_level_tuple,
 OuterSequence outer_sequence, const Tuple& t,
 Variables& vars, Args&&... args) {

  operation op;
  op = process_block(std::integer_sequence<int, Next, Rest...>{},
    function_level_tuple, append_sequence<N>(outer_sequence),
    get_function<N>(t), vars, std::forward<Args>(args)...);

  if (op == operation::_done) {
    return if_else_ < (N < get_size_v<Tuple> - 1)>([&](auto& t) {
    return process_block(std::integer_sequence<int, N + 1>{},
```

RECALL OUR PROCESS BLOCK FUNCTION



```
operation op;
op = process_block(std::integer_sequence<int, Next, Rest...>{},
  function_level_tuple, append_sequence<N>(outer_sequence),
  get_function<N>(t), vars, std::forward<Args>(args)...);

if (op == operation::_done) {
  return if_else_ < (N < get_size_v<Tuple> - 1)>([&](auto& t) {
  return process_block(std::integer_sequence<int, N + 1>{},
      function_level_tuple, outer_sequence, t, vars);
    },
    [] {return operation::_done; }, t);
 }
 return op;
}
```

ADDED TO SUPPORT WHILE IN PROCESS BLOCK

```
op = ...

using continue_sequence = std::integer_sequence<int, 0>;

op = if_else_<is_while_t<std::decay_t<decltype(get_function<N>(t))>>::value>(
  [&](auto& t) {
    operation lop = op;
    while (lop == operation::_continue) {
      lop = process_block(continue_sequence{}, function_level_tuple,
        append_sequence<N>(outer_sequence), get_function<N>(t), vars);
    }
    if (lop == operation::_break) { lop = operation::_done; }
    return lop;
  }, [&](auto&) {return op; }, t);

if (op == operation::_done) {
```

PROCESS WHILE

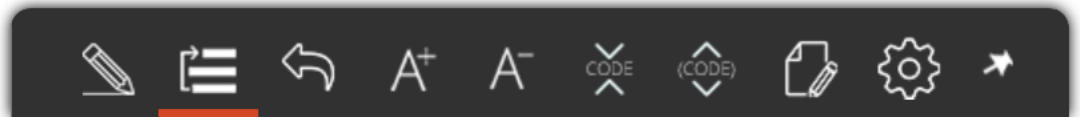
```
template<int N, class FunctionLevelTuple, class OuterSequence,
 class Tuple, class Variables, class... Args>
static operation process(type2type<while_type>, FunctionLevelTuple& function_level_tuple,
  OuterSequence outer_sequence, const Tuple& t, Variables& vars, Args&&... args) {

  process_outermost_block(append_sequence<0>(append_sequence<N>(outer_sequence)),
    function_level_tuple, vars, std::forward<Args>(args)...);
  return operation::_suspend;
}
```

MAKE WHILE TRUE

```
template <class... T> auto make_while_true(T &&... t) {
  const auto dummy_while_terminator = [](auto &, auto &) {
      return operation::_continue;
  };
  auto tuple =
      std::make_tuple(std::forward<T>(t)..., dummy_while_terminator);
  return detail::while_t<decltype(tuple)>{ std::move(tuple) };
};
```

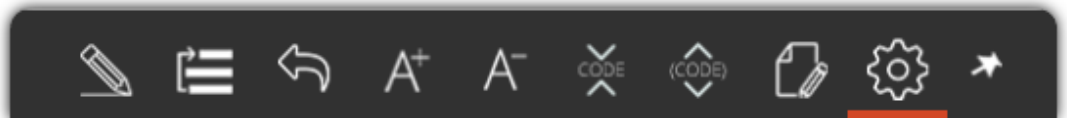
NOW THIS WORKS

```
make_block(stackless_coroutine::make_while_true(
          [](auto &context, auto &variables) {
            variables.socket.async_read_some(buffer, context);
            return context.do_async();
          },
          [](auto &context, auto &variables, auto &ec, auto length) {
            if (ec) return context.do_async_break();
            boost::asio::async_write(socket,buffer,context);
            return context.do_async();
          },
          [](auto &context, auto &variables, auto &ec, auto) {
            if (ec) return context.do_break();
            return context.do_next();
          }));
```



IF

```
template<class F,class Tuple, int Else>
struct if_t {
    F f;
    Tuple t;
    enum{else_start = Else};
};
```



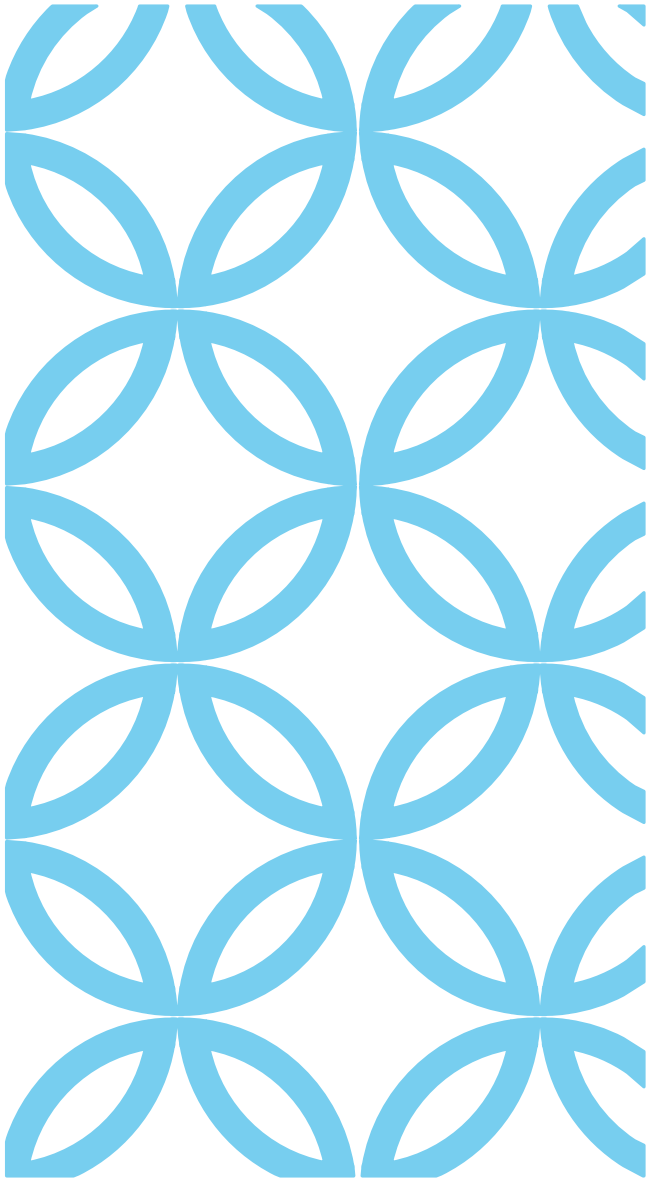
MAKE IF

```
template<class F, class T1, class T2>
auto make_if(F&& f, T1&& t1, T2&& t2) {
  auto dummy_done = [](auto&, auto&) {return operation::_done; };

  auto combined = std::tuple_cat(std::forward<T1>(t1), std::make_tuple(dummy_done),
    std::forward<T2>(t2), std::make_tuple(dummy_done));

  using combined_type = decltype(combined);

  return detail::if_t<std::decay_t<F>,  combined_type, std::tuple_size<T1>::value + 1>
    {std::forward<F>(f), std::move(combined)};
}
```

PROCESSING IF

```
template<int N, class FunctionLevelTuple, class OuterSequence,
class Tuple, class Variables, class... Args>
static operation process(type2type<if_type>, FunctionLevelTuple& function_level_tuple,
  OuterSequence outer_sequence, const Tuple& t, Variables& vars, Args&&... args) {

  const auto& ift = get_function<N>(t);
  using ift_type = std::decay_t<decltype(ift)>;
  operation op;
  if (ift.f(vars)) { op = process_block(std::integer_sequence<int, 0>{},
            function_level_tuple, append_sequence<N>(outer_sequence), ift.t, vars);
  }
  else {  op = process_block(std::integer_sequence<int, ift_type::else_start >{},
       function_level_tuple, append_sequence<N>(outer_sequence), ift.t, vars);
  }
  if (op == operation::_done) { op = operation::_next; }
  return process_next<N>(op, function_level_tuple, outer_sequence, t, vars);
}
```

```
auto block = make_block(stackless_coroutine::make_while(
    [](auto &variables) { return variables.begin != variables.end; },
    [](auto &context, auto &variables) {
      return context.do_async_yield(variables.begin->path().string());
    },
    stackless_coroutine::make_if(
        [](auto &variables) {
          return filesystem::is_directory(variables.begin->path());
        },
        std::make_tuple(
            [](auto &context, auto &v) {
              v.gen = make_recursive_directory_generator(v.begin->path());
            },
            stackless_coroutine::make_while(
                [](auto &variables) {
                  return variables.gen.begin() != variables.gen.end();
                },
                [](auto &context, auto &variables) {
                  return context.do_async_yield(
                      std::move(*variables.gen.begin()));
                },
                [](auto &context, auto &variables) {
                  ++variables.gen.begin();
                })),
        std::make_tuple()),
    [](auto &context, auto &variables) { ++variables.begin; }));
return make_generator<string, variables_t>(block,p);
```



https://github.com/jbandela/stackless_coroutine/tree/channel_dev

C++14

No macros

No shared_ptr

Header-only, < 700 lines of code

Works with and without exceptions

No dependencies

Drop in replacement for callbacks

Generators

Boost Software License

APPETIZER: BLOCKS, ASYNC, WHILE, AND IF