Asynchronous Operations

Dietmar Kühl
Bloomberg L.P.
ACCU 2015

Copyright Notice

The Problem

- make progress during long running operations
  - external source/sink like I/O or database
  - involved computations run elsewhere
- do not block dedicated threads (like UI thread)
- I/O will be used in examples

Simple Approach

- just read without consideration of concurrency:

  ```
  int size = read(fd, buffer, sizeof(buffer));
  ```

- may block indefinitely if there is no data, yet
- even for files may need to wait for data to arrive

Threads

- run long operations in a thread:

```
future<int> f = async(read, fd, buffer, size);
// …
int size = f.get(); // caution: will probably block
```

- f.wait_for() == future_status::ready to test
- still blocks one thread, just not the current one

Blocking Threads

- threads are intended to *run* code, not to block
- threads are fairly heavy weight:
  - each thread allocates a stack
  - the system tracks which threads can run
  - number of threads is relatively limited

I/O Thread(s)

- run I/O operations on dedicated threads:
  - auto f = io.read(fd, buffer, sizeof(buffer))
- instead of blocking I/O operations poll(2):
  - few (e.g., one) threads are blocked on I/O
  - much better resource use

Implementing read()

- send request to another thread:

```
std::future<int> io::read(int fd, void* b, size_t s) {
   std::promise<int> p;
   std::future<int>     rc(p.get_future());
   this->enqueue(std::move(p), ::read, fd, b, s);
   return rc;
}
```

I/O Thread

- use poll() (or similar) to block on many streams:

```
std::vector<pollfd> polls(fill_from_requests());
if (0 < poll(polls.data(), polls.size(), 1)) {
    int index = find_next_entry(polls);
    request& r = requests[index];
    r.promise.set_value(::read(r.fd, r.b, r.s));
}
```

Blocking std::future<T>

- non-ready std::future<T> block upon access:

```
auto f = io.read(fd, buffer, sizeof(buffer));
// do some work
auto size = f.get(); // may still block
use(buffer, size);
```

- worse: you can’t easily check if that will block

std::future<T>::then()

- proposed std::future<T>::then() (N3857):

```
auto f0 = io.read(fd, buffer, sizeof(buffer));
auto f1 = f0.then([=](std::future<int> size){
                      return use(buffer, size.get()); }));
```

- register a function to be executed next

- called with ready future: size.get() won’t block but it can also communicate an error

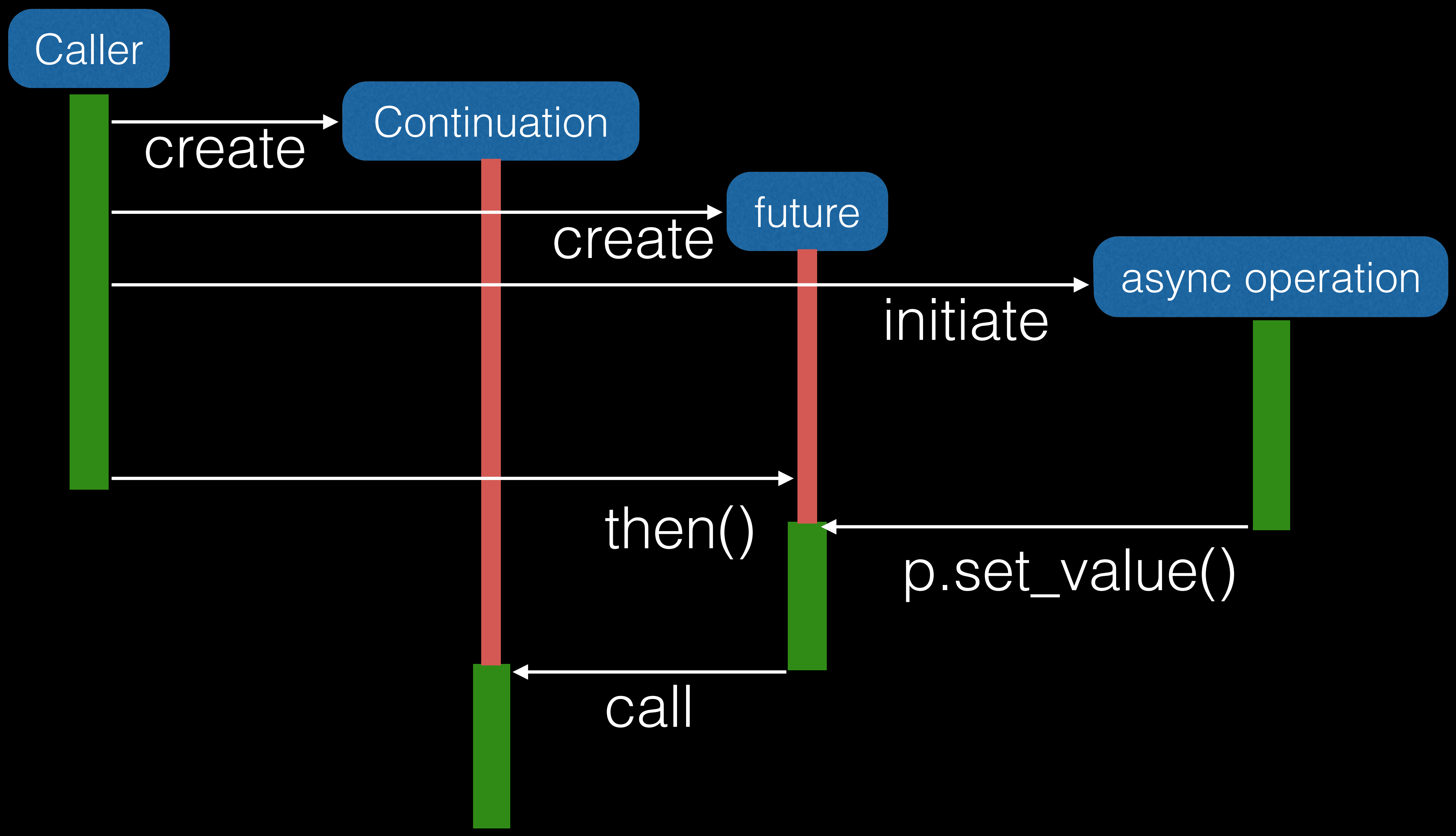
std::future Improvements

- then(), also using launch policy and executor
- unwrap(): get inner future from a nested future (std::future<std::future<T>>)
- f.is_ready(): determine if f.get() won't block
- when_any(), when_any_swapped(), when_all(): yield std::future<std::vector<std::future<T>>>
- make_ready_future<T>(value)

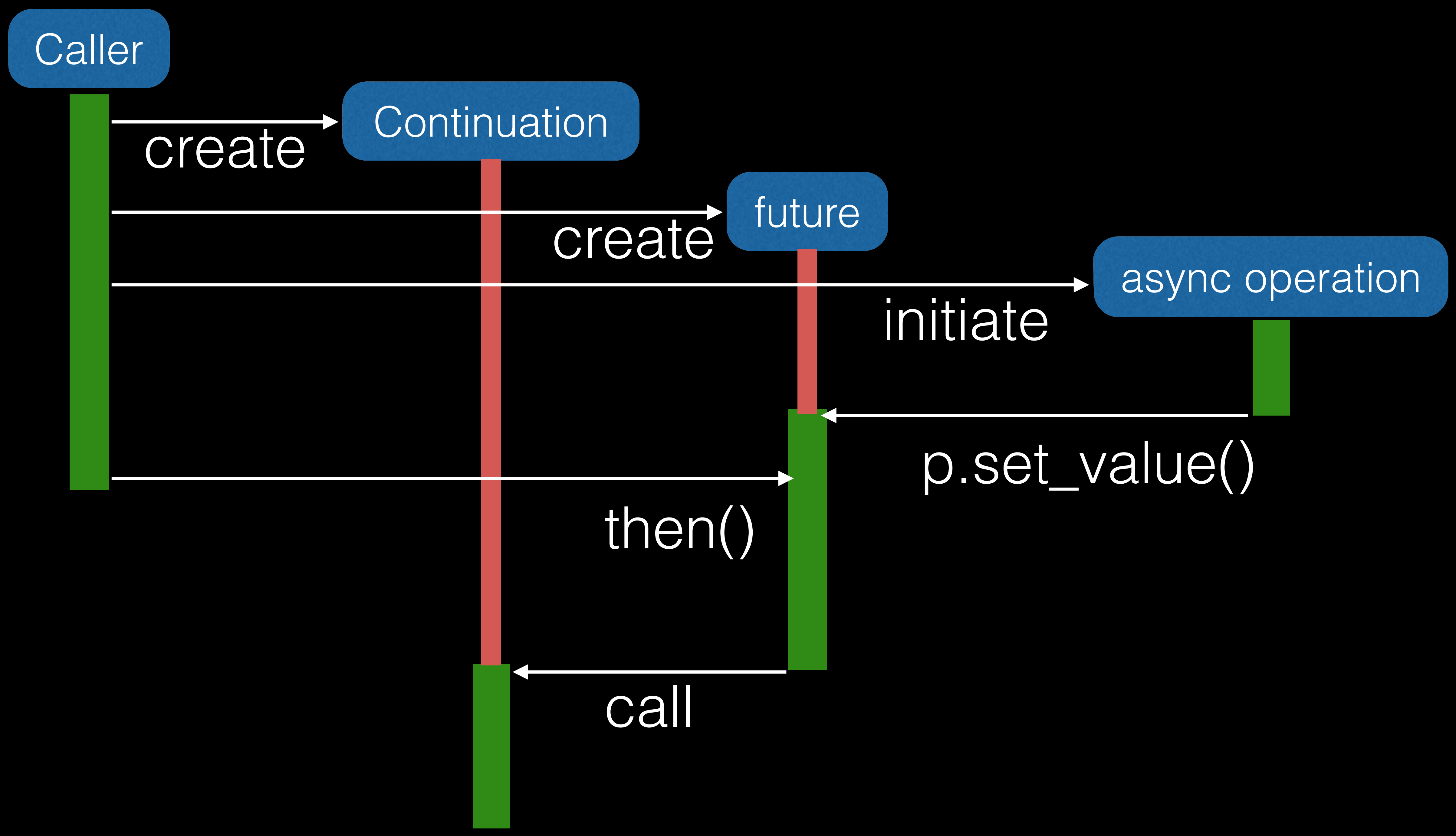
Continuation Functions

- async operations: inversion of control:
  - instead of blocking specify how to carry on
  - well-known approach: event-driven
  - sadly, it is relatively complex (see later though)
- std::future<T>::then() requires synchronisation

.then() Synchronization



.then() Synchronization



Allowing Callbacks

- control gets inverted anyway: allow callbacks

```
io.read(fd, buffer, sizeof(buffer),
        [=](int size){ use(buffer, size); });
```

- pass executor for control of running callbacks
- *can* use .then(): set up before adding request

No Threads!

- callbacks called when work is available
- central blocking place, e.g., poll()
  - *can* have multiple poll()ing threads
- one thread ⇒ no synchronisation needed
- trade-off: sequence replaced by function calls

Callbacks

```
void run() {
    socket.async_read_some(asio::buffer(buffer),
        [=](asio::error_code ec, size_t size) {
            if (!ec) on_read(size);
        });
}
void on_read(size_t size) {
    // …
    run();
}
```

Completion Token

- different strategies for continuations are useful
- specification on how processing should proceed
    - callback ⇒ continue when ready
    - use_future ⇒ get back a suitable future
    - use form of coroutine ⇒ continue function

How to Complete

```
template<class CT>
auto async_xyz(A… a, CT&& token) {
    completion_handler_t
        <decay_t<CT>, void(R… r)>
      ch(forward<CT>(token));
    async_result<decltype(ch)> result(ch);
    trigger asynchronous xyz => calling ch
    return result.get();
}
```

Completion: Callbacks

- default type for the handler is the token
- async_result<C>
  - calls the callback upon completion
  - returns void from result.get()

Get a Future

```
void run() {
    auto f = s.async_read_some(asio::buffer(b),
                               asio::use_future);
    f.then([=](asio::error_code ec, size_t size) {
        if (!ec) on_read(size);
}
void on_read(size_t size) {
    // …
    run();
}
```

Completion: Future

- use of future indicated by a token: use_future
- use_future handler uses a promise/future:
    - completion function sets the promise
    - result.get() returns the future

Coroutines

- functions are single-entry, single-exit
- coroutines are
  - started once
  - suspended/reentered multiple times
  - exited once

Coroutines in C++

- multiple proposals
  - resumable functions (N4402)
  - stackless coroutines (N4453)
  - stackful coroutines (N4397; sort of)
  - unified stackless/stackful coroutines (N4398)

Stackful Coroutines

- can stop/resume from any statement
- stores stack up to entry point
    - relatively memory intensive
    - split stacks do help
    - at least 2 pages are typically needed

Stackful Example

```
using C=coroutines::asymmetric_coroutine<int>;
C::pull_type pull([](C::push_type& push) {
  for (int i(0); i != 2; push(i++))
    std::cout << "yielding i=" << i << '\n';
  });

  std::cout << "created pull-type\n";
  for (; pull; pull())
    std::cout << "pulled " << pull.get() << '\n';
}
```

Stackful Output

```
yielding i=0
created pull-type
pulled 0
yielding i=1
pulled 1
```

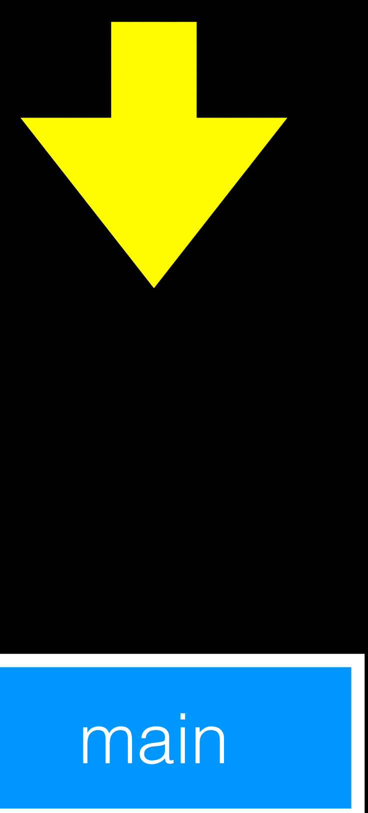
Program Behaviour

program start

initial stack

Program Behaviour

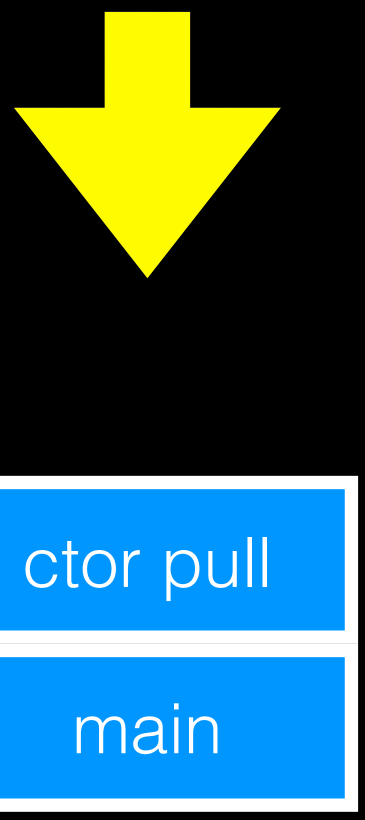
main()



initial stack

Program Behaviour

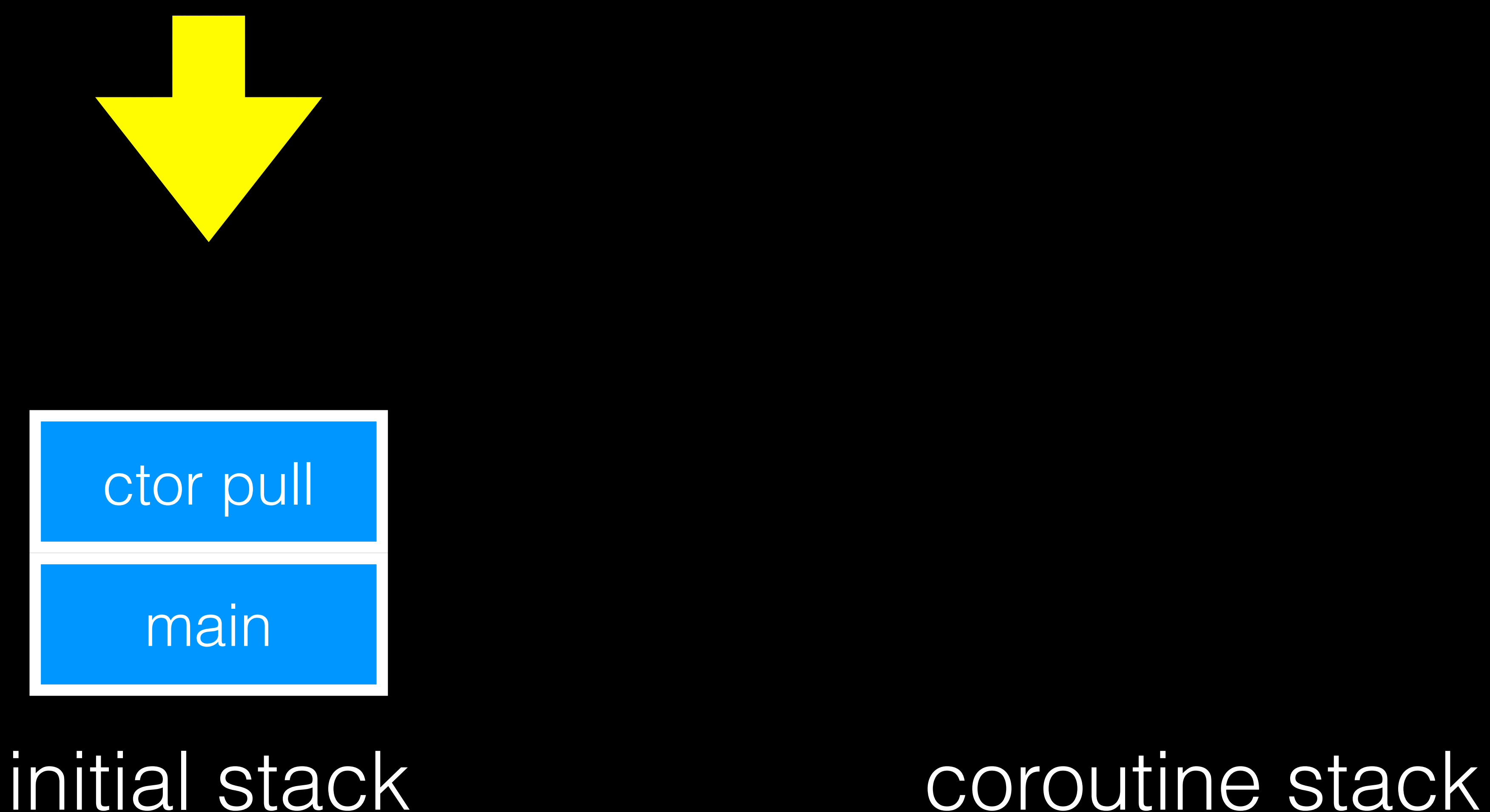
create: C::pull_type



initial stack

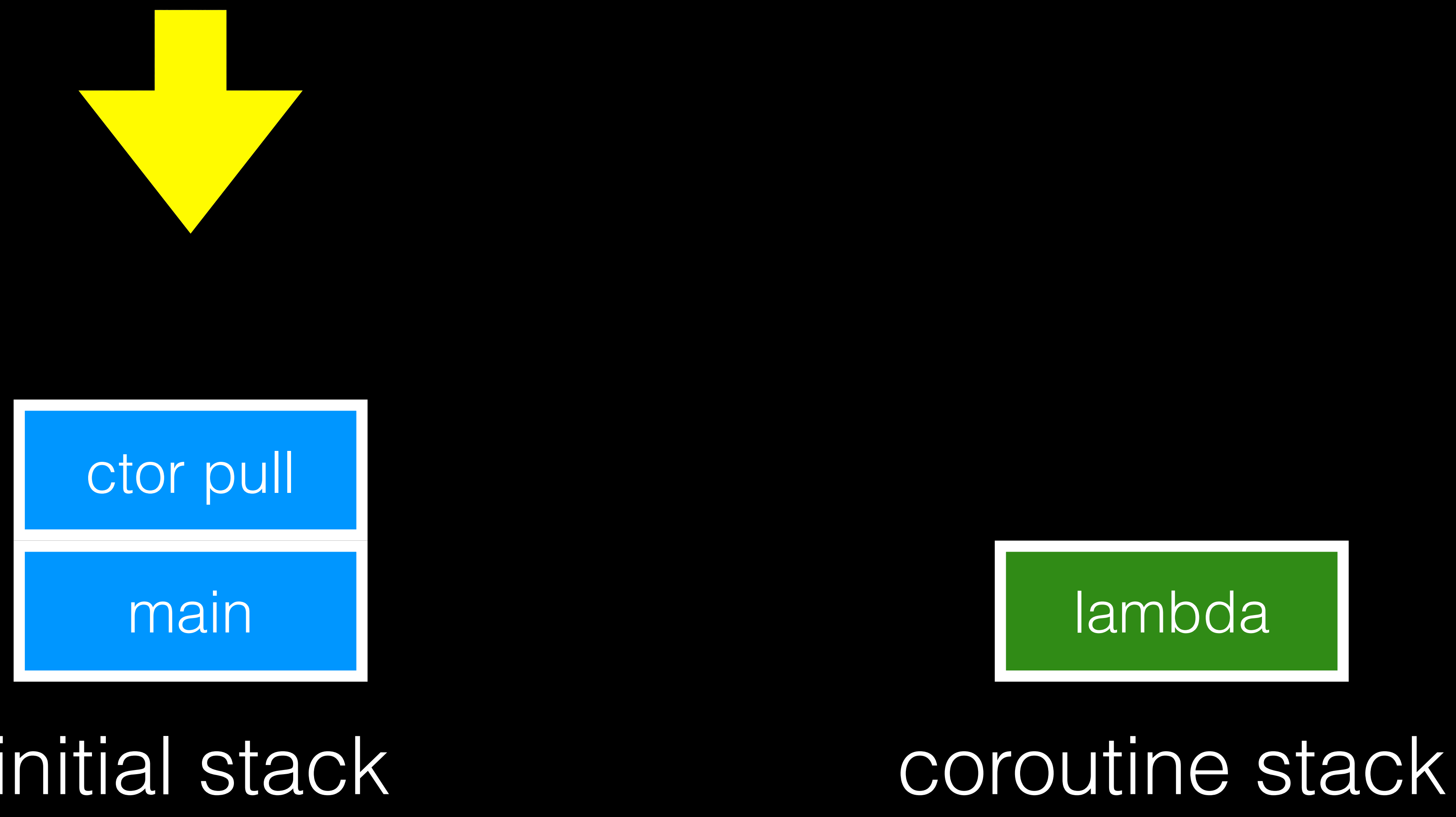
Program Behaviour

create: C::pull_type - allocate coroutine stack



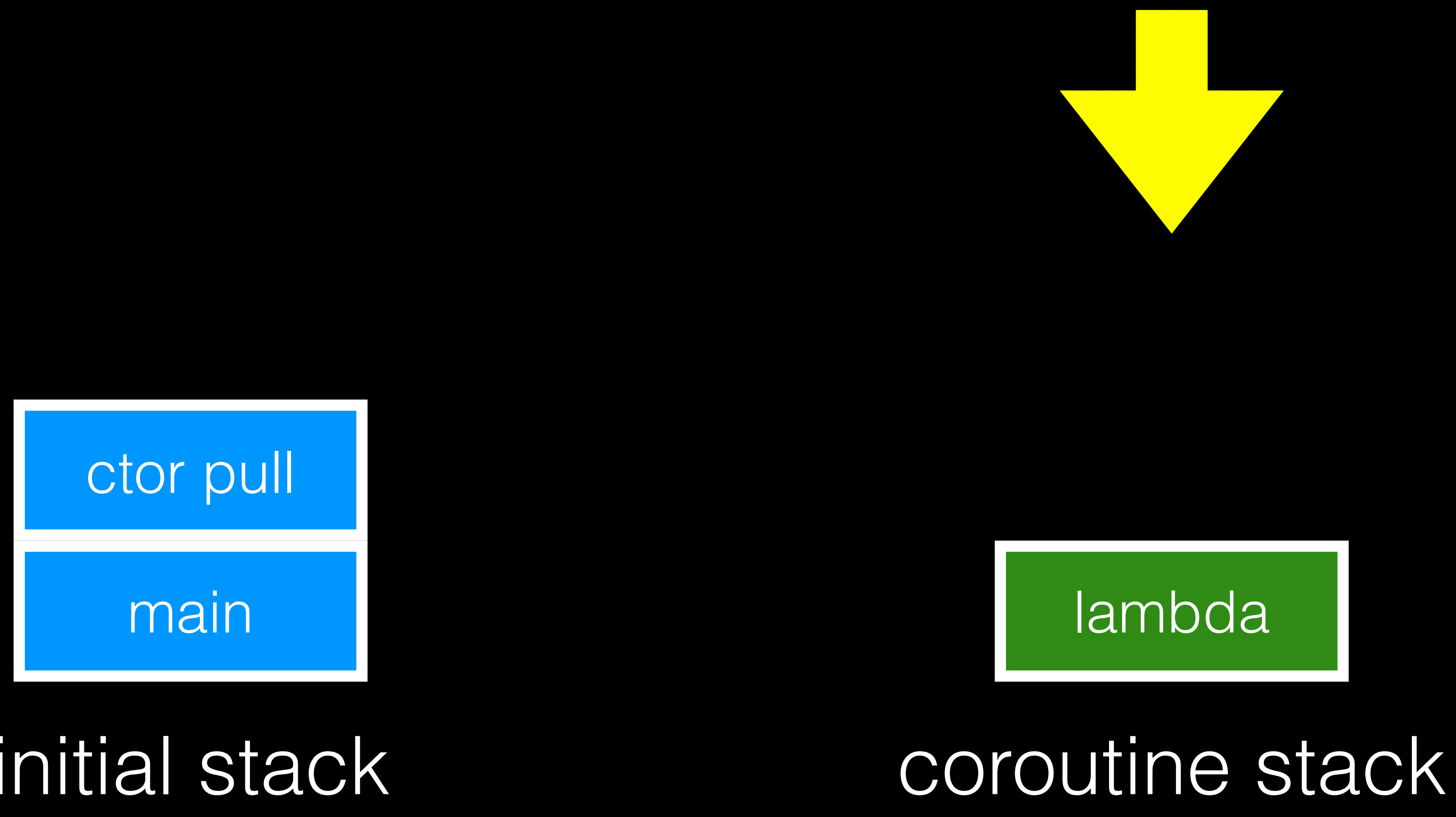
Program Behaviour

create: C::pull_type - create lambda function



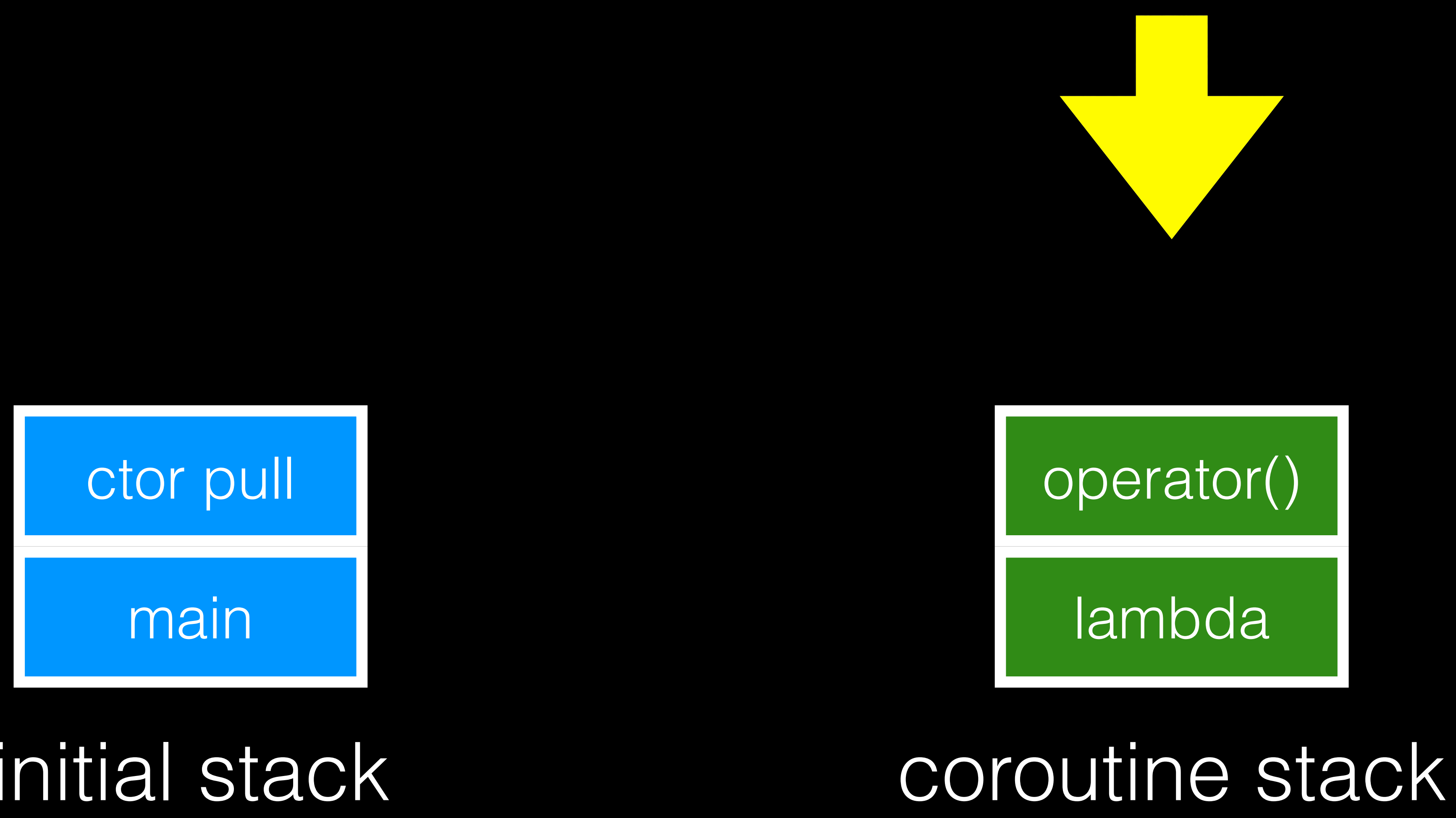
Program Behaviour

create: C::pull_type - yield



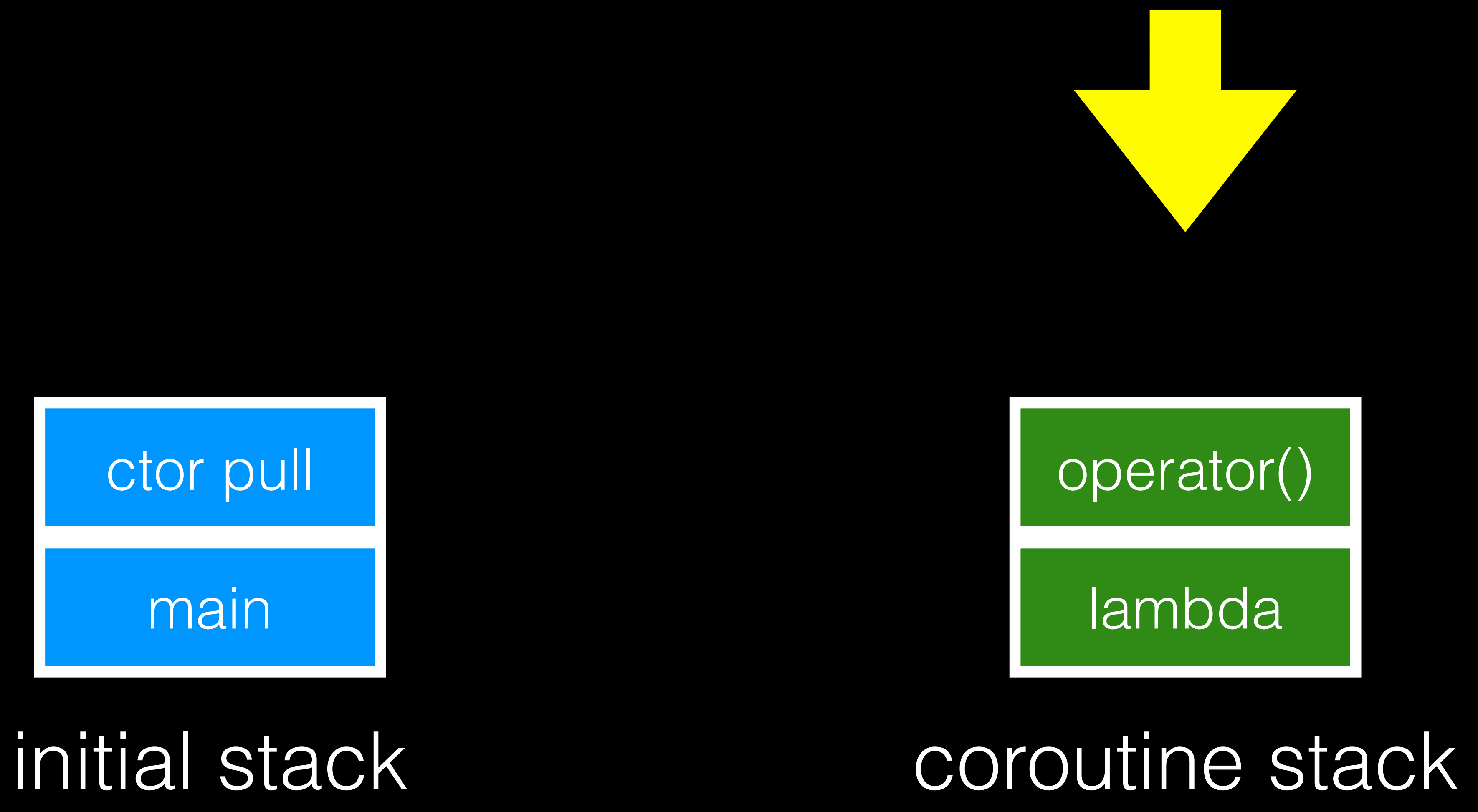
Program Behaviour

run lambda function



Program Behaviour

for (int i = 0; i != 2;



Program Behaviour

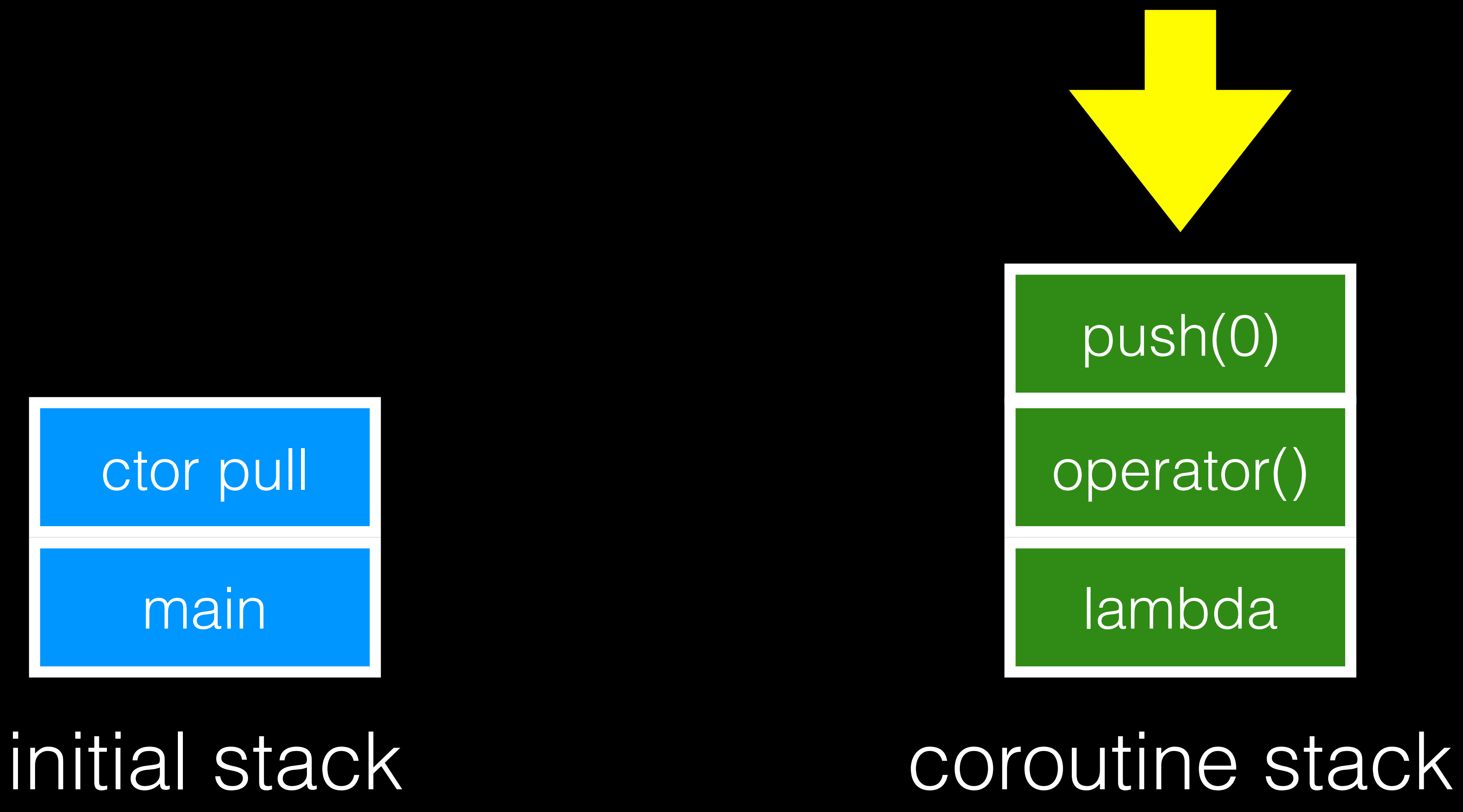
std::cout << “yielding i=“ << 0 << ‘\n’;

initial stack

coroutine stack

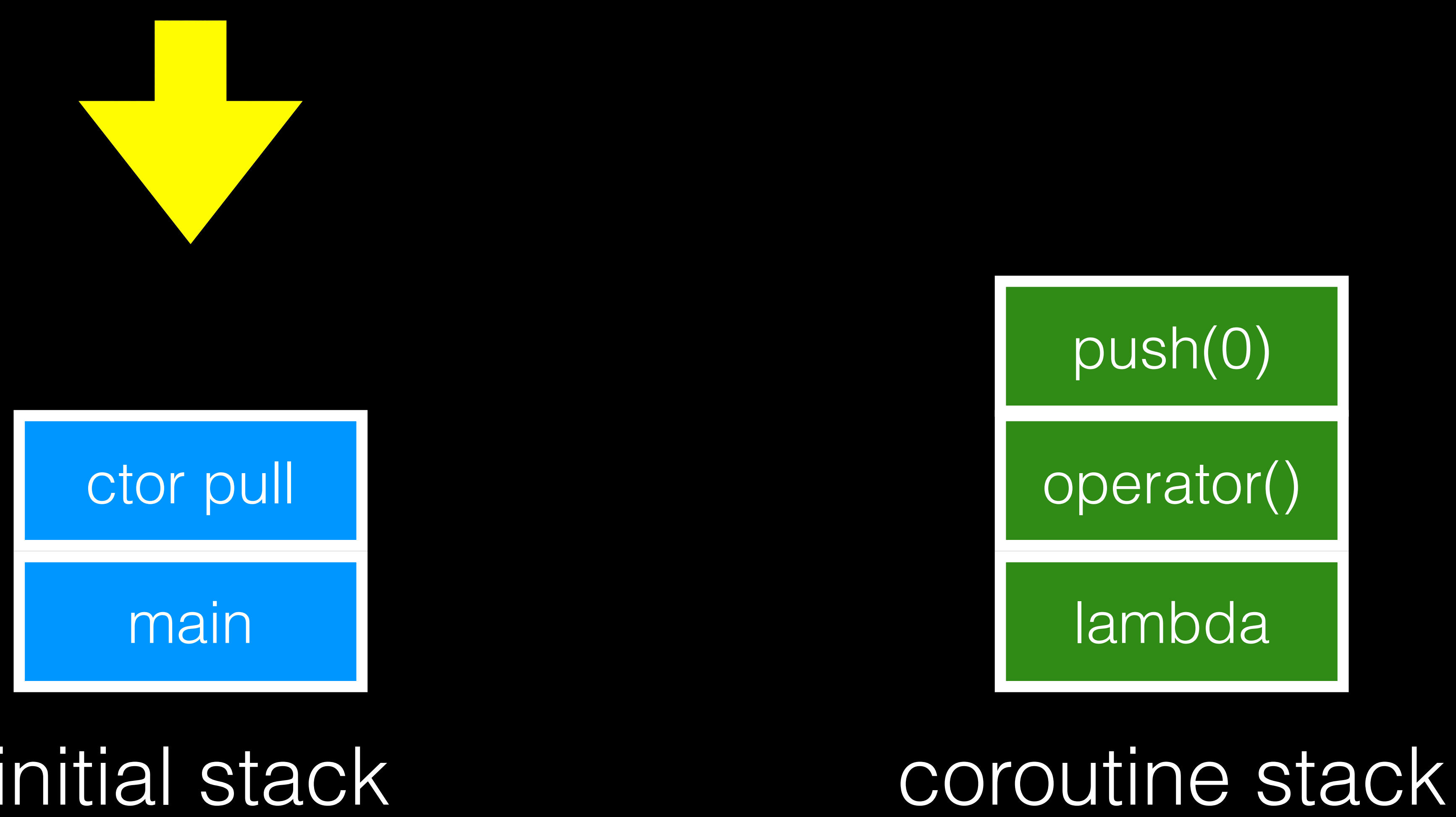
Program Behaviour

for (….; push(0)) - set value



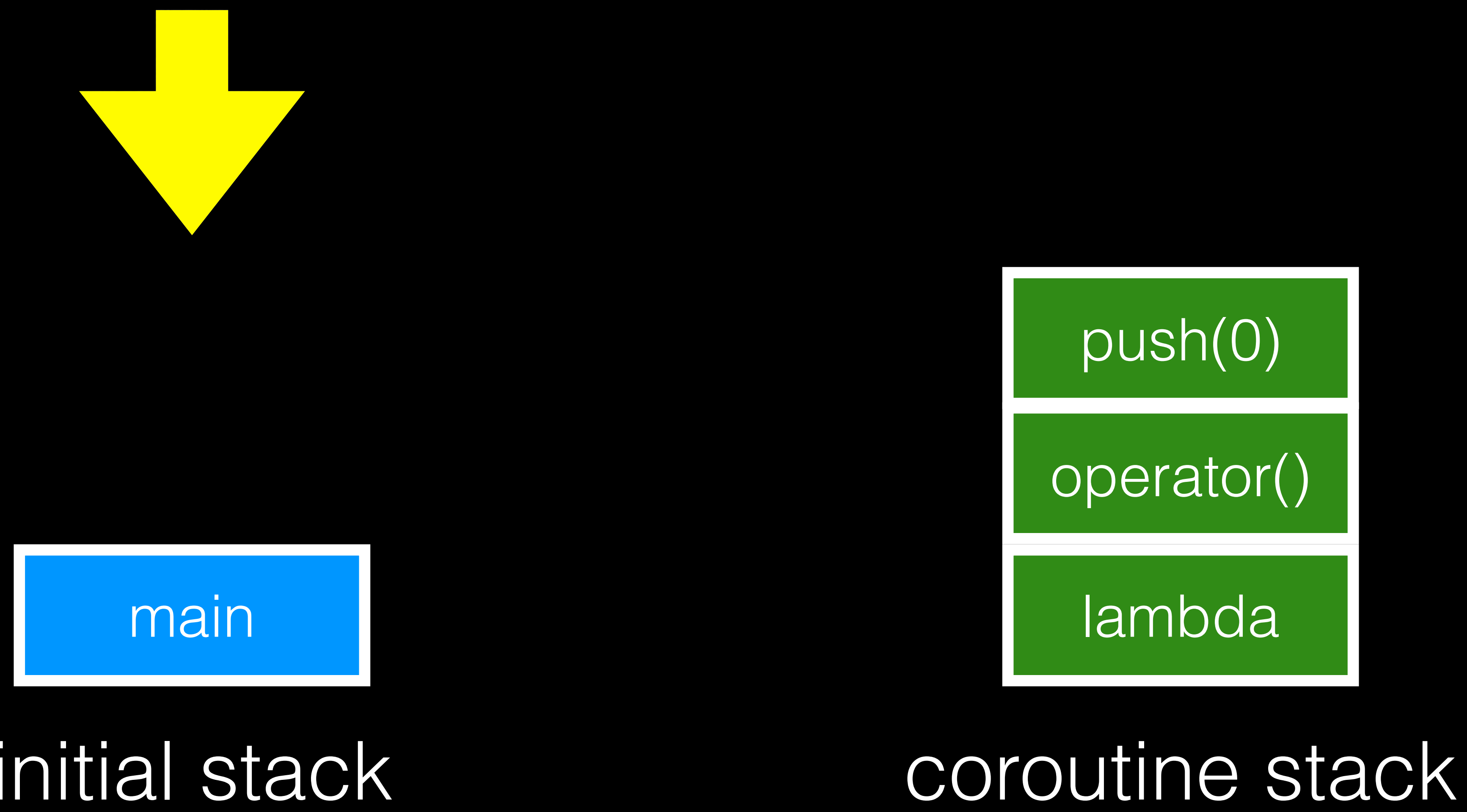
Program Behaviour

push(0); - yield



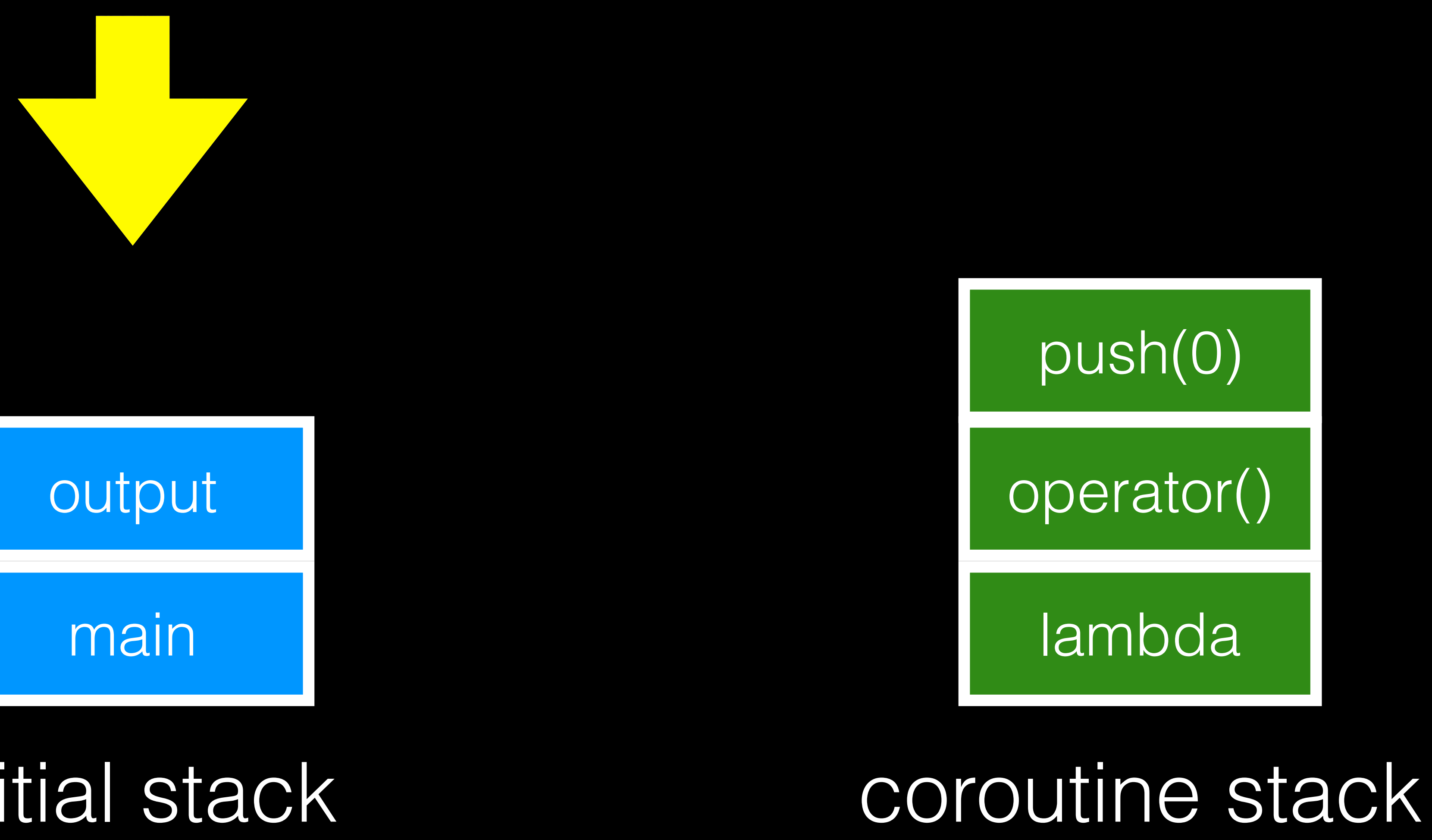
Program Behaviour

create: C::pull_type - finish



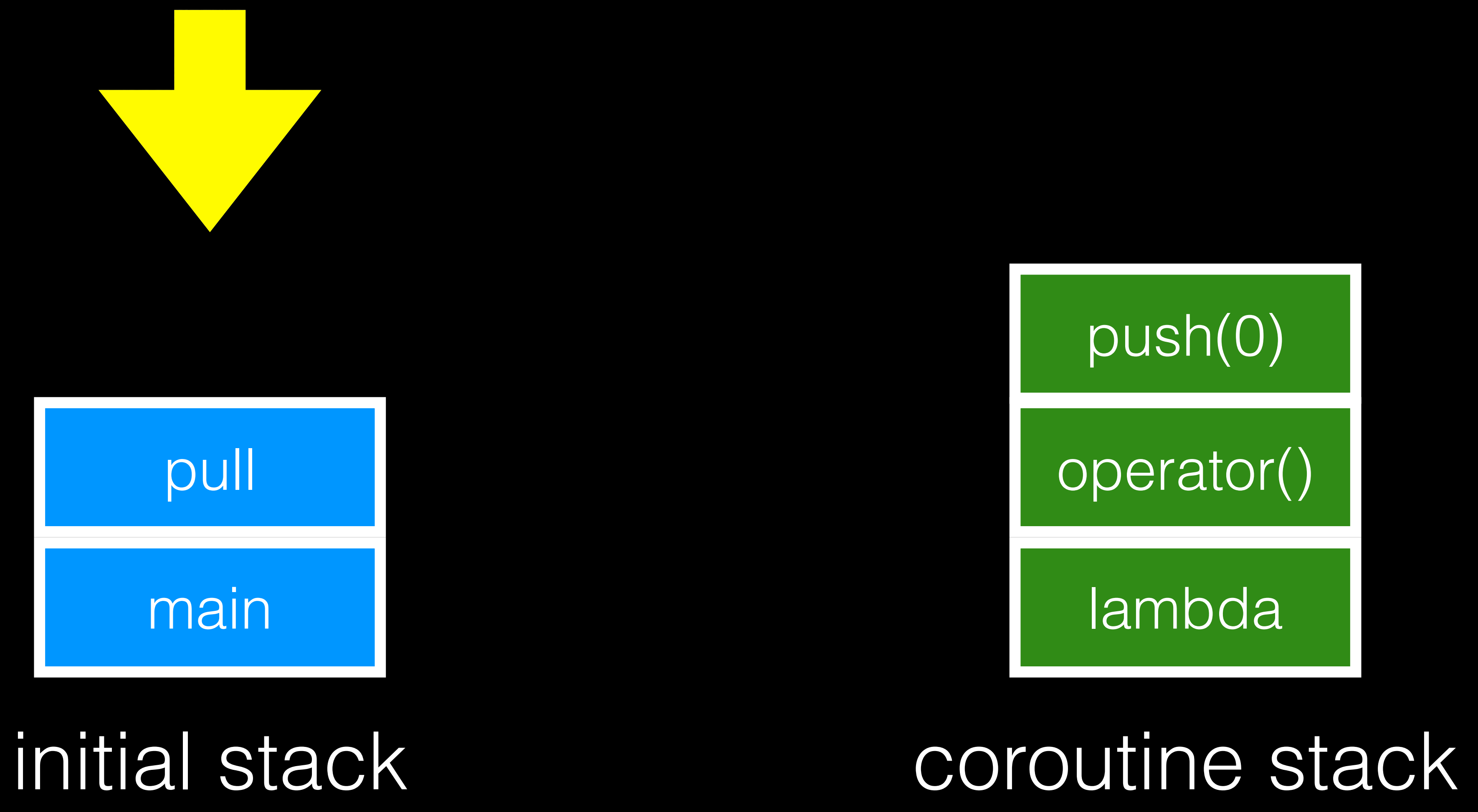
Program Behaviour

```
std::cout << "created pull-type\n";
```



Program Behaviour

for (; pull;



Program Behaviour

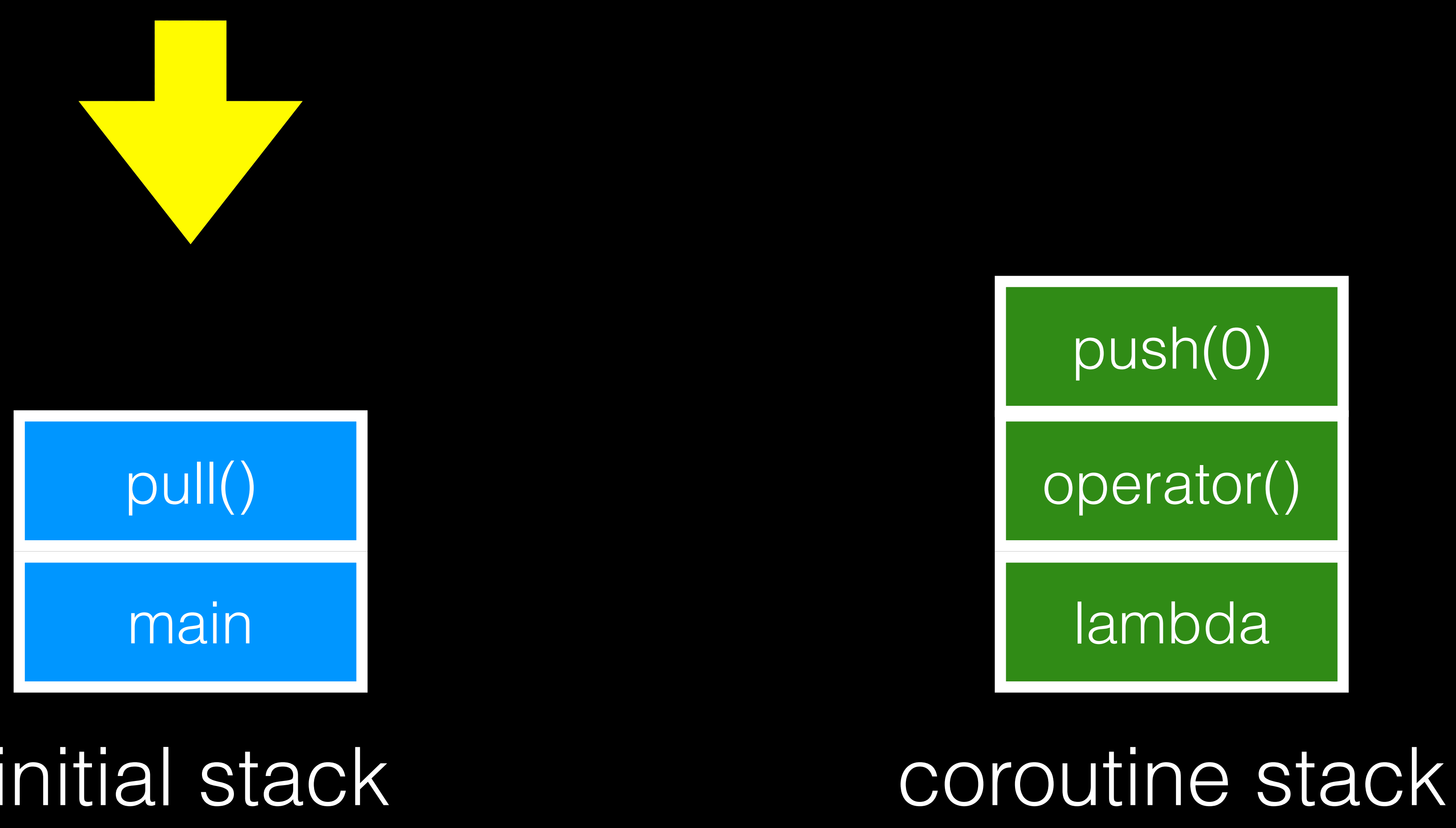
std::cout << “pulled: “ << pull.get() << ‘\n’;

initial stack

coroutine stack

Program Behaviour

for (…; pull())



Program Behaviour

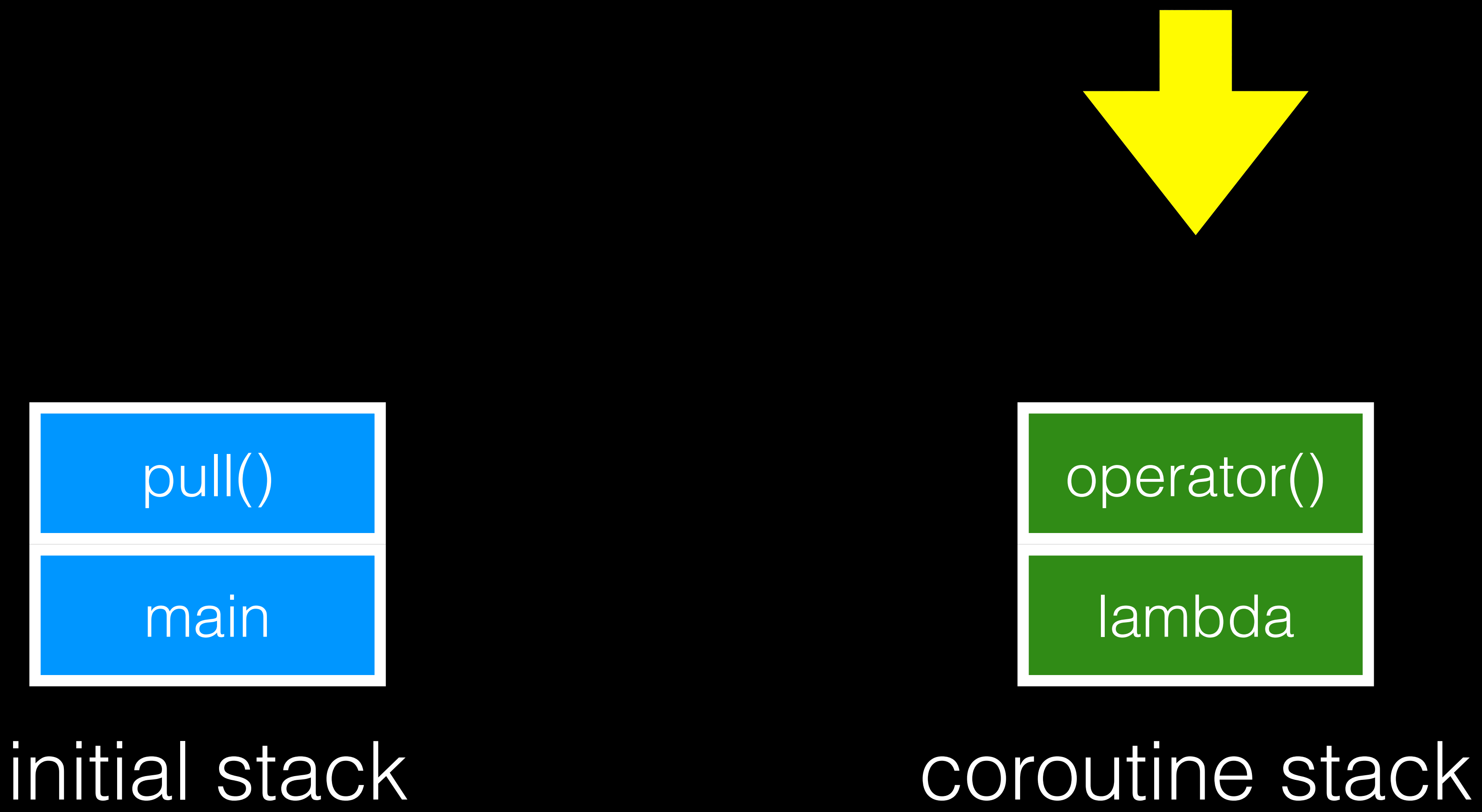
for (…; pull()) - yield

initial stack

coroutine stack

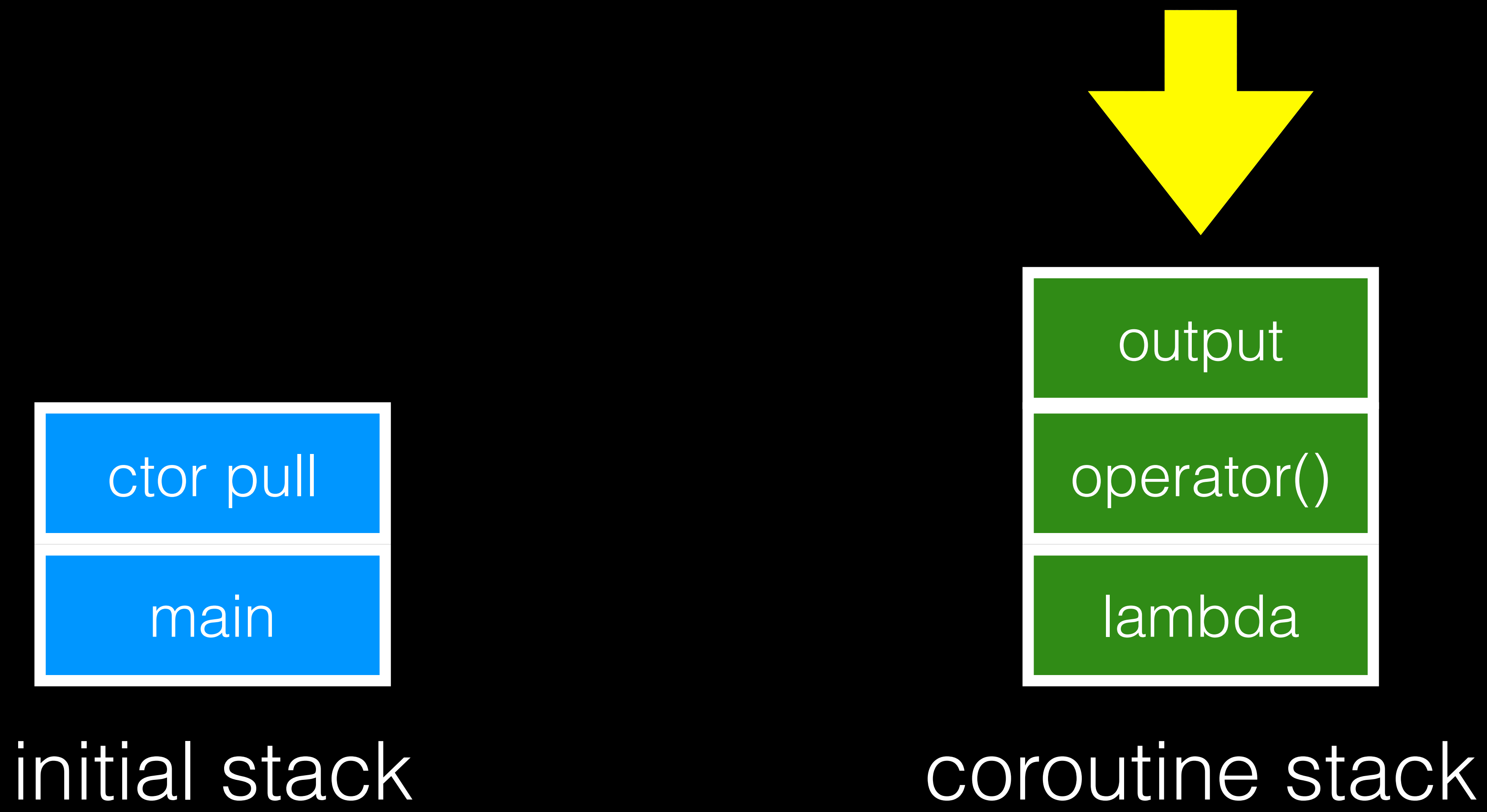
Program Behaviour

for (…; i != 2; i++)



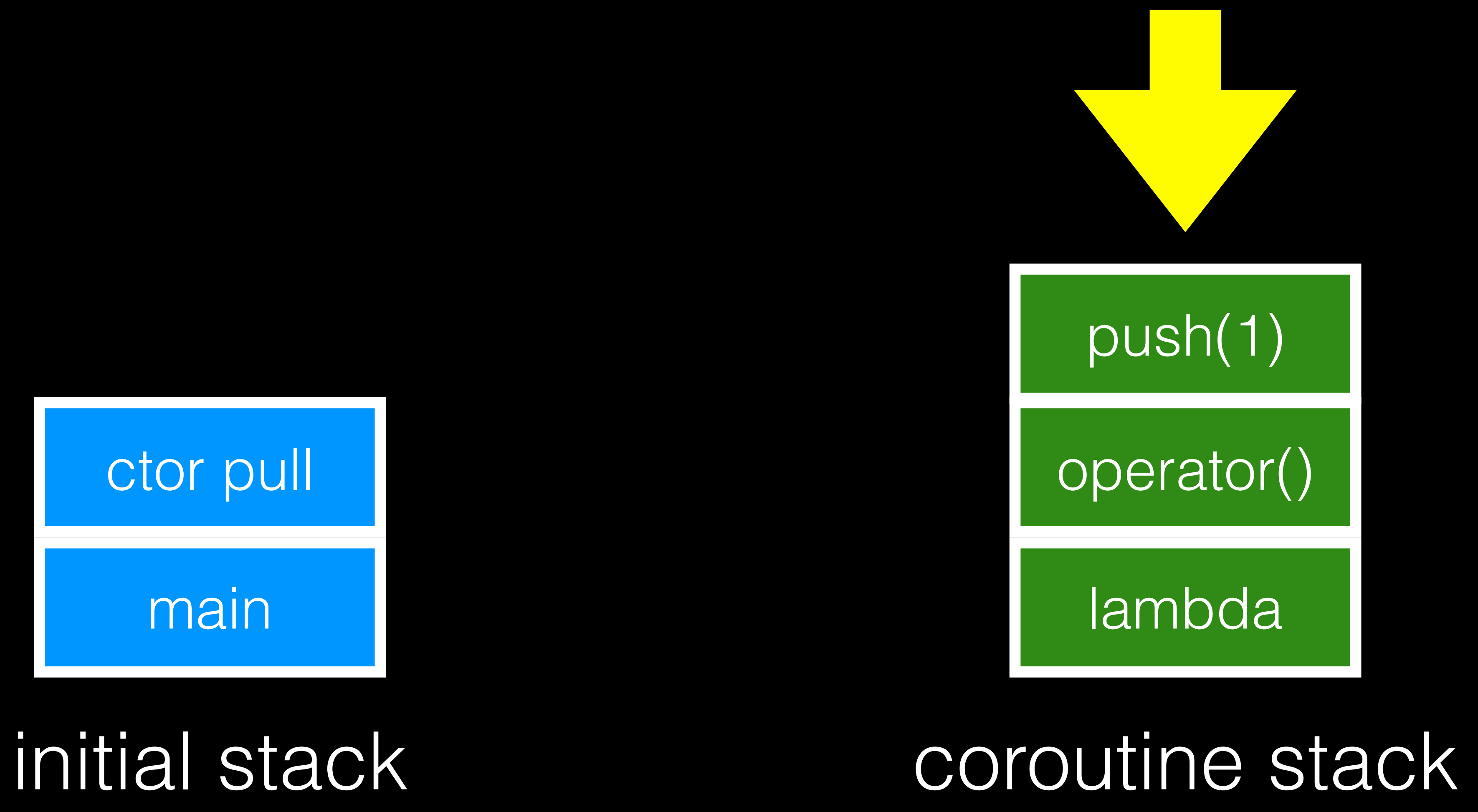
Program Behaviour

std::cout << “yielding i=“ << 1 << ‘\n’;



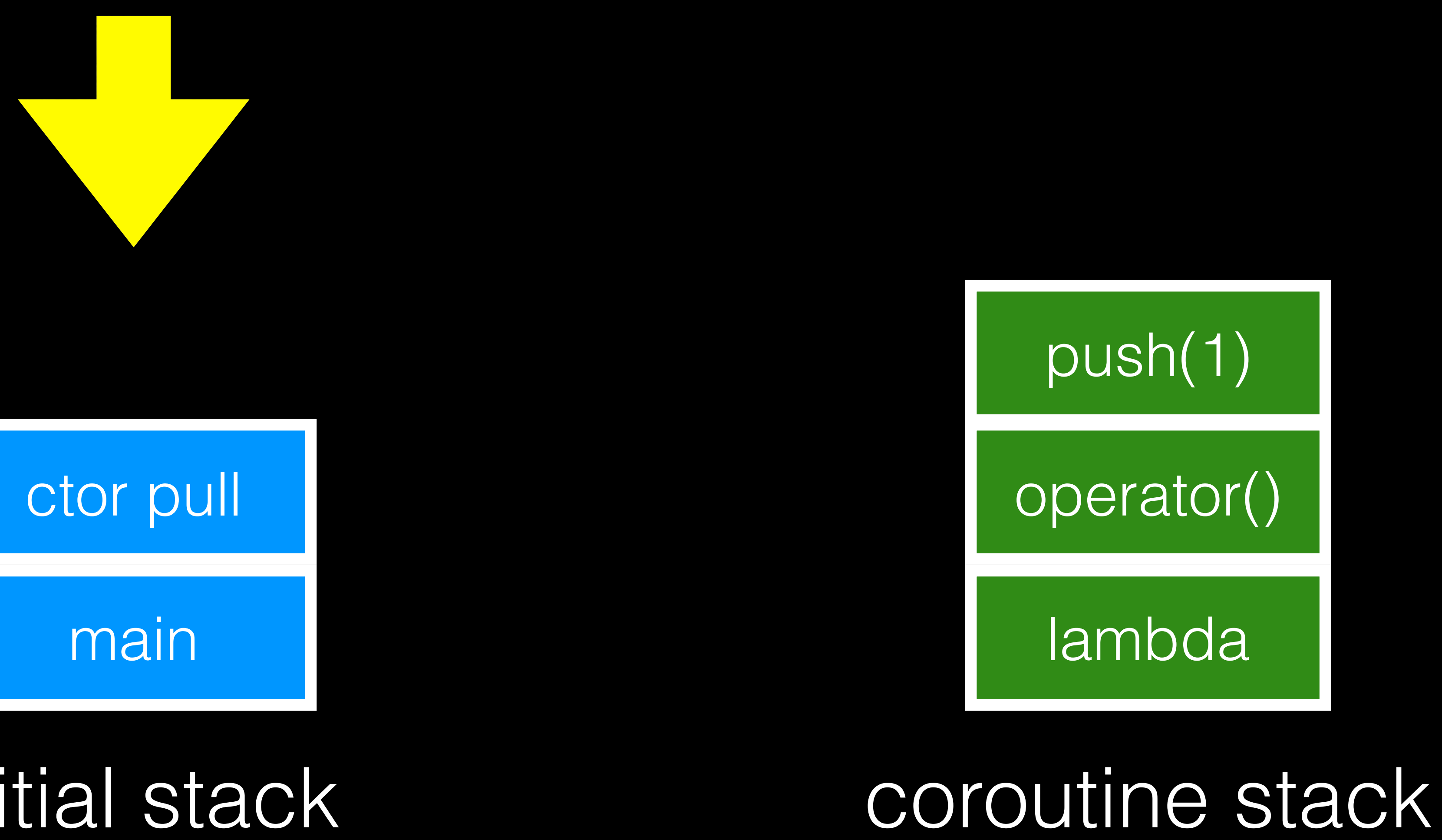
Program Behaviour

for (….; push(1)) - set value



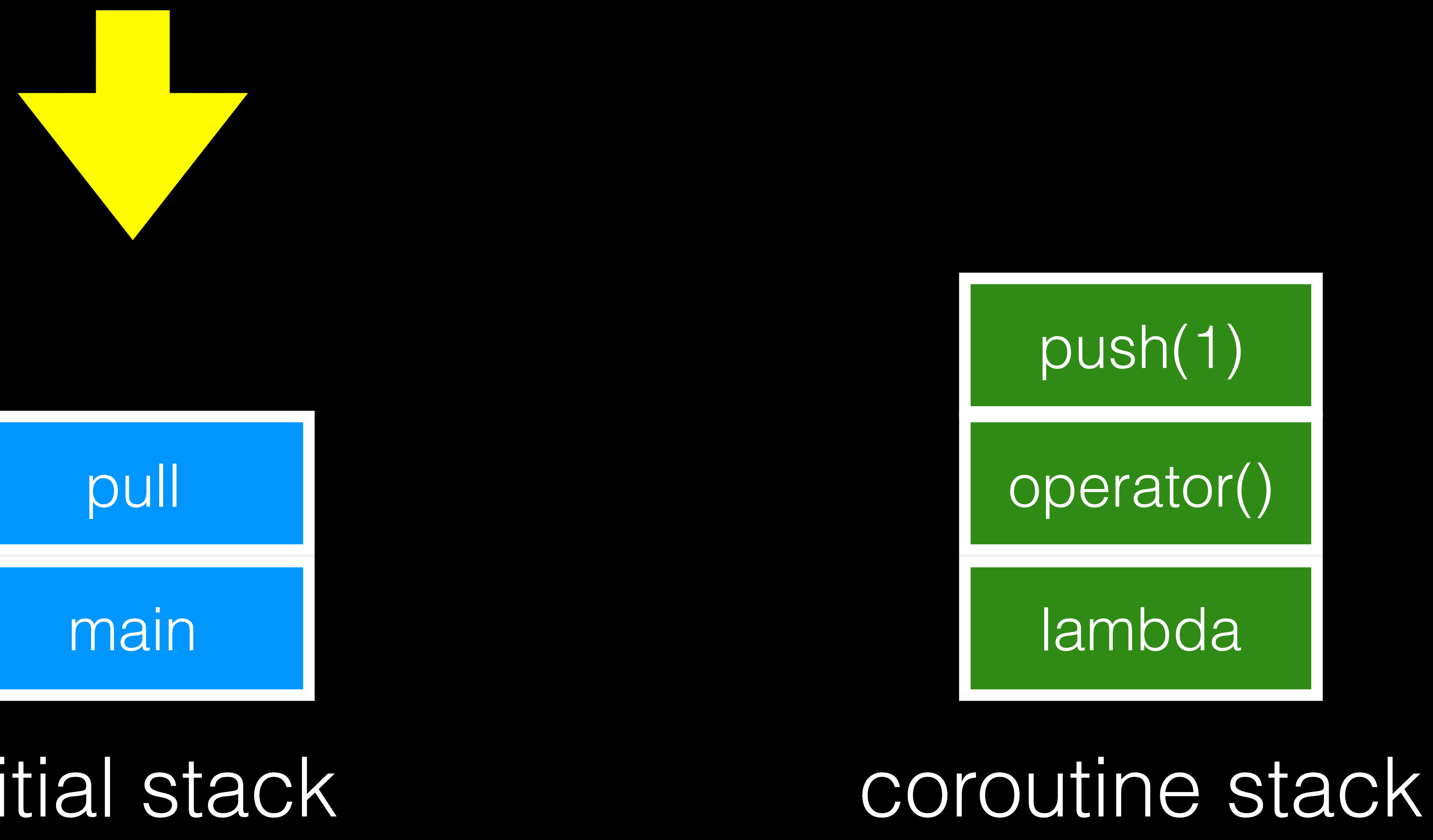
Program Behaviour

push(1); - yield



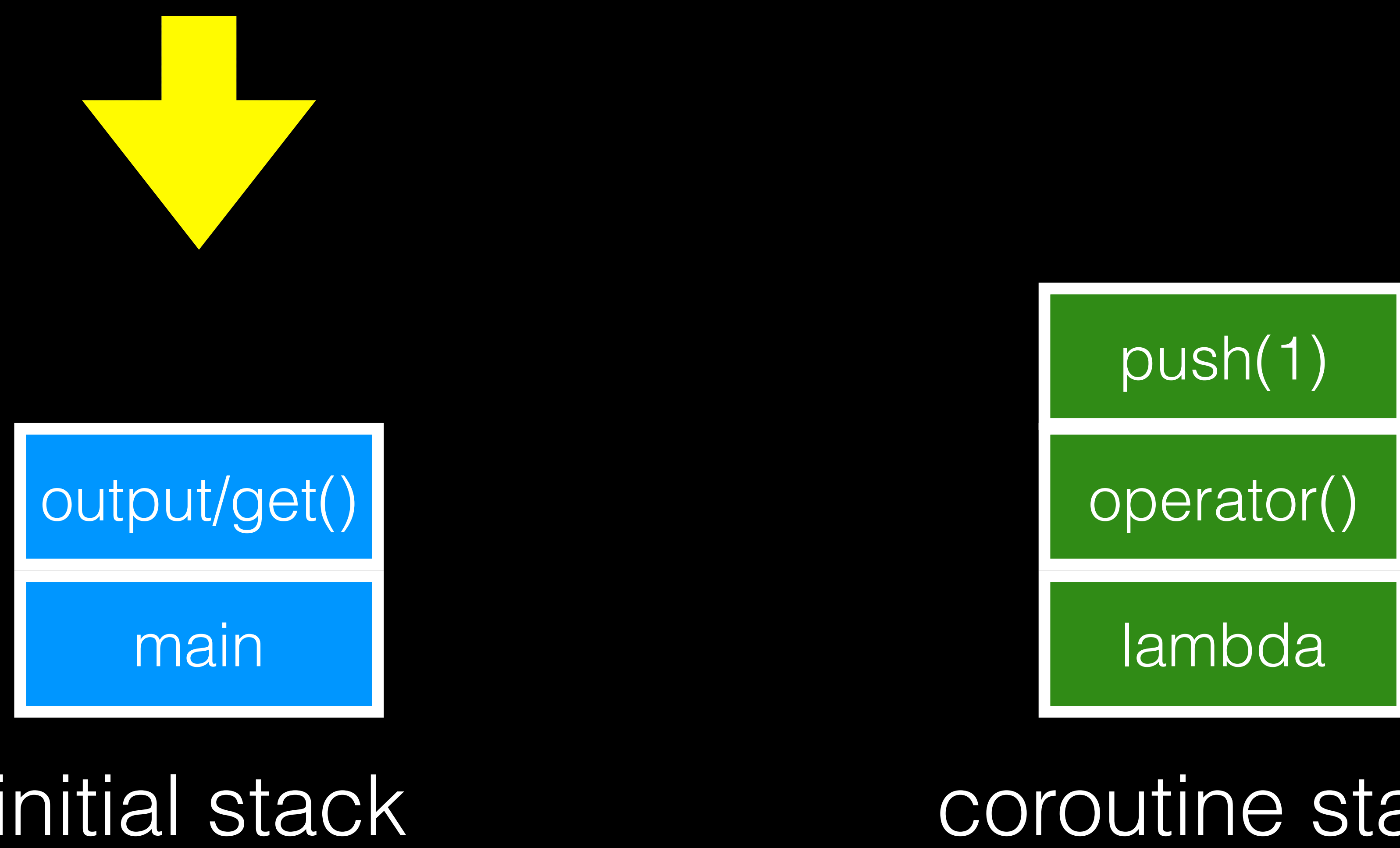
Program Behaviour

for (… ; pull;)

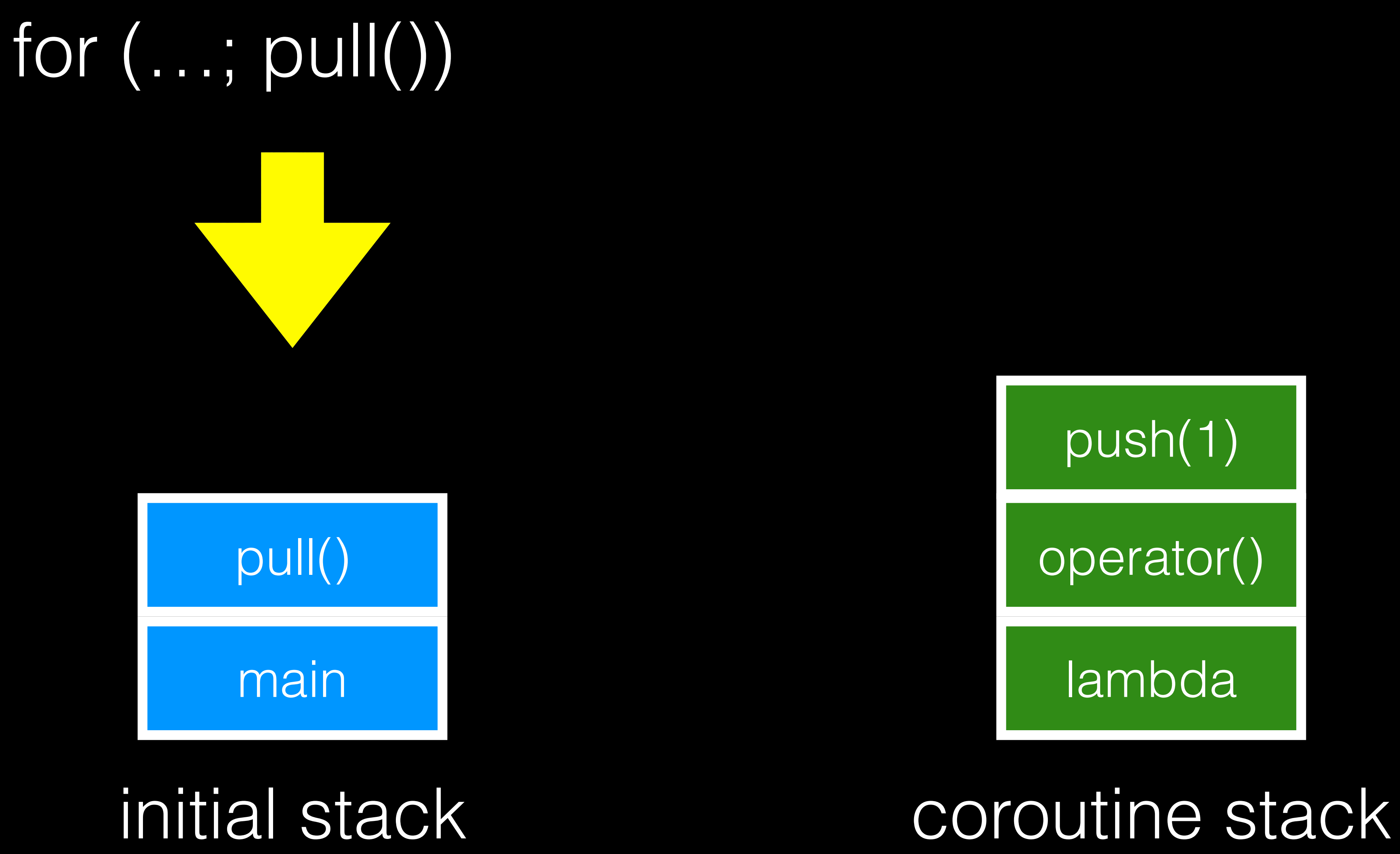


Program Behaviour

```
std::cout << “pulled: “ << pull.get() << ‘\n’;
```



Program Behaviour



Program Behaviour

for (…; pull()) - yield

initial stack

coroutine stack

Program Behaviour

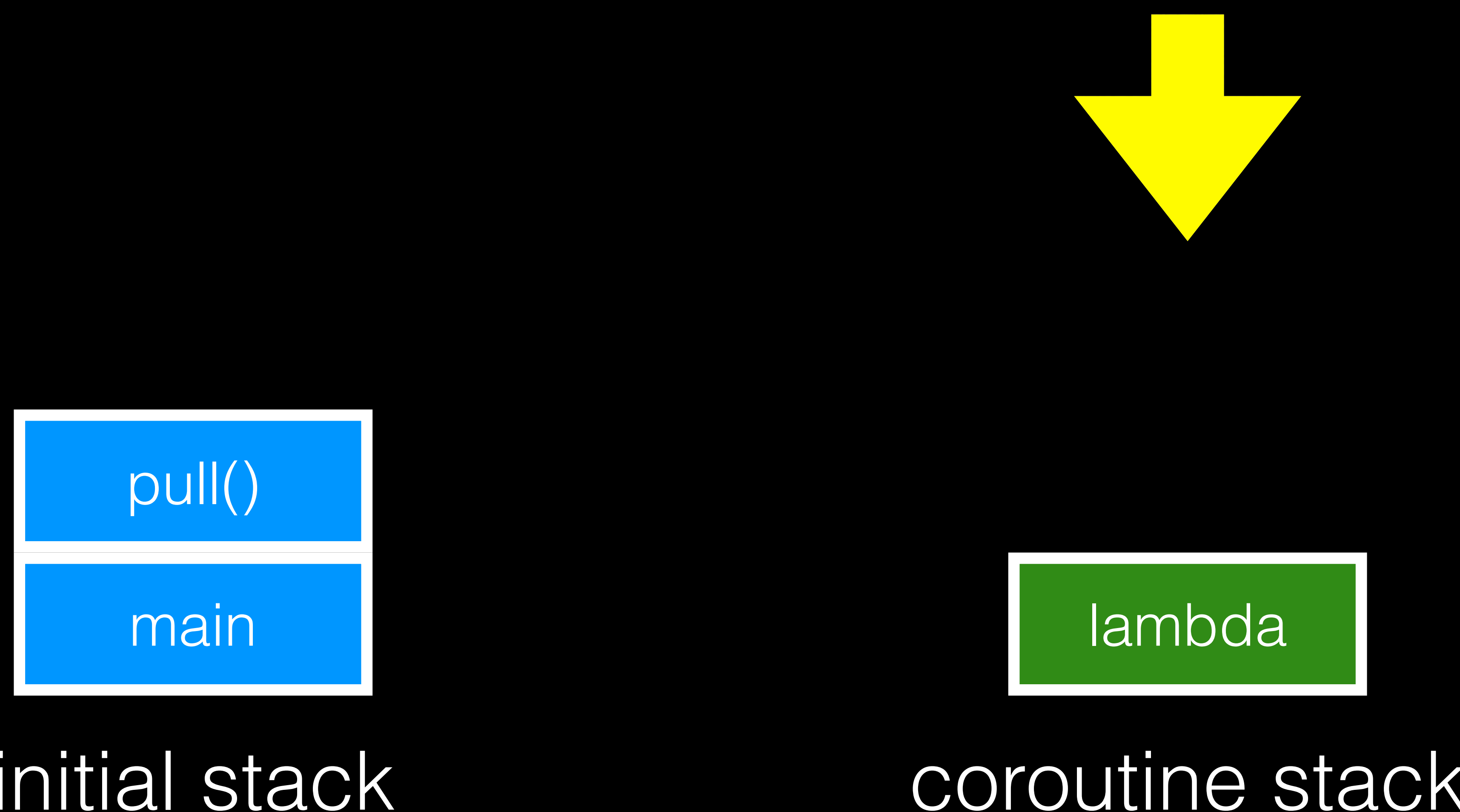
for (…; i != 2; i++)

initial stack

coroutine stack

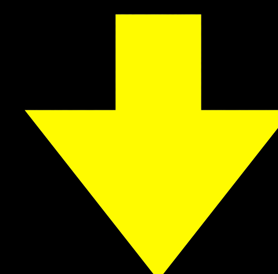
Program Behaviour

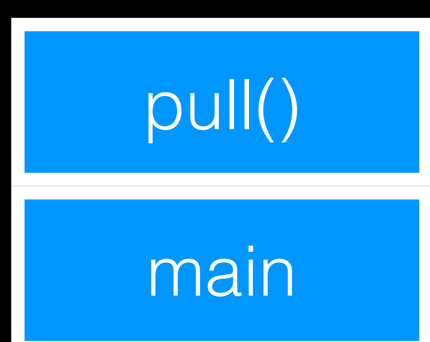
return from operator()



Program Behaviour

destroy lambda



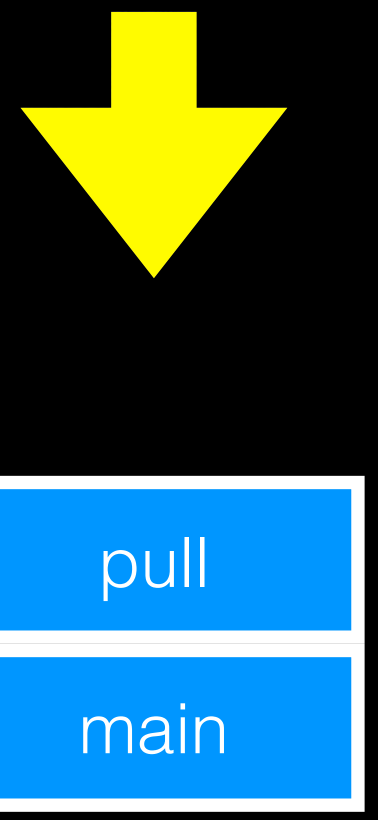


initial stack

coroutine stack

Program Behaviour

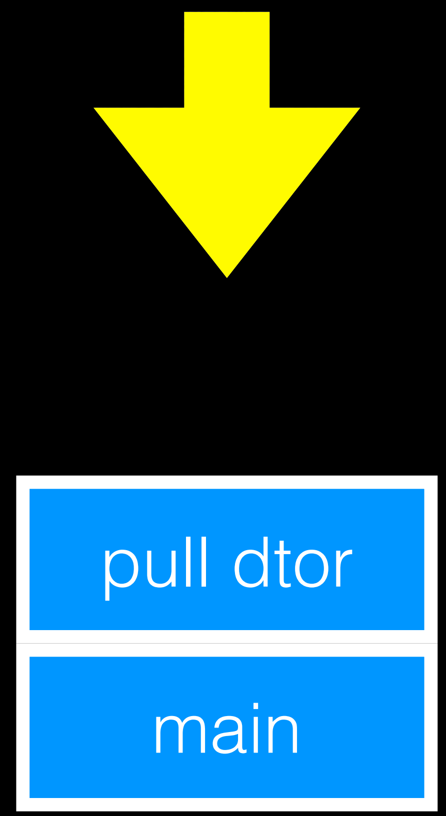
for (… ; pull;)



initial stack

Program Behaviour

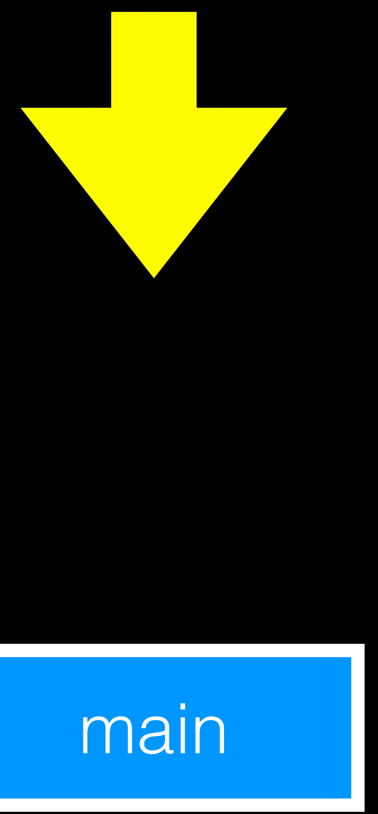
destroy pull object



initial stack

Program Behaviour

finish main()



initial stack

Symmetric/Asymmetric

- asymmetric: context yielded to is always implicit
    - initiator: pulling (not a coroutine)
    - coroutine: pushing
- symmetric: context yielded to is specified
    - look more flexible
    - contexts need more management

Async vs. Coroutine

```
void run(socket& s, yield_context yield) {
    char buffer[1024];
    size sz = s.async_read_some(asio::buffer(b),
                                              yield);
    // use sz and buffer
}
```

Completion: Stackful

- use of coroutine via yield_context object
- handler uses push context for completion
- arguments become elements of the return
- result.get() calls pull() and returns pull.get()

Stackless Coroutines

- cannot suspend from nested function calls
- minimal state: an int where to carry on
- any local variable kept while being resumable
- very little state ⇒ there can be many instances
- quite fast to suspend/resume
- can be tested to see if they can be resumed

Stackless Example

```
struct function : asio::coroutine {
    int operator()() {
        reenter(*this) {
            yield return 17;
            yield return 19;
        }
        return 23;
    }
};
```

Stackless Use

- the example uses a macro hack
- … but can be used straight forward

```
function fun;

while (!fun.is_complete()) {
    std::cout << fun() << '\n';
}
```

Async vs. Stackless

```
struct function : asio::coroutine {
    std::shared_ptr<rep> rep;
    void operator()(error_code ec = error_code(),
                    size_t size = 0) {
        if (!error) reenter(*this) for (;;) {
            yield rep->socket.async_read_some(
                asio::buffer(rep->buffer), *this));
            use(rep->buffer, size);
        }
    }
};
```

Completion: Stackless

- completion token is just a function object
- same behaviour as for callbacks:
- async_result<C>
    - calls the callback upon completion
    - returns void from result.get()

Executing Completions

- some thread needs to run completion handlers
  - run on the thread completing the operation
  - run somewhere else
- approach: use executor to schedule handler
- determined from involved objects

Executor

- schedules tasks (nullary function objects)
- different ways to schedule tasks:
  - ex.dispatch(fun, alloc): maybe immediately
  - ex.post(fun, alloc): after post() but ASAP
  - ex.defer(fun, alloc): after defer() but not ASAP
- use an execution context for the actual work

io_service

- execution context capable of doing I/O work
- inactive unless at least one thread is running it
  - implements a pool of threads
  - ios.run() to add current thread to the pool
  - multiple threads can join the pool
  - one thread ⇒ serial processing

Strand

- executor limiting execution to one thread
  - tasks are not executed concurrently
  - independent tasks are synchronised
- order of tasks being added is retained

Fiber

- execution policy processing on one thread
- cooperative/non-preemptive scheduling
- uses similar techniques as coroutines
- fiber-versions of classes used with threads:
  - mutex, condition_variable, future, promise

Standardization

- implemented in boost and separately
- networking TS for ASIO (n4478)
- different models for
  - executors
  - coroutines and resumable functions

Conclusion

- asynchronous scheduling allows concurrency
- may use one thread avoiding many problems
- coroutines ease the use of callbacks

Questions?