



GENERATORS, COROUTINES AND OTHER BRAIN UNROLLING SWEETNESS

ADI SHAVIT

@adishavit :: videocortex.io

CppCon :: 2019



+

GENERATORS, COROUTINES AND OTHER BRAIN UNROLLING SWEETNESS

+

ADI SHAVIT

@adishavit :: videocortex.io

CppCon :: 2019

FUNCTIONS A.K.A. SUB-ROUTINES



- Let's iterate!
- One function:
 1. Iterates
 2. Operates

```
void vectorate(std::vector<int> const& v)
{
    for (auto e: v)           // 1. iterate
        std::cout << e << '\n'; // 2. do something: print e
}
```

1947 •

- What if we need another operation?
 - Sum?
 - Both: Sum + Print?

DID YOU KNOW?

The concept of a function, or **sub-routine** goes back to one of the first computers, the ENIAC, in the late 1940s and the term **sub-routine** is from the early 1950s.

FUNCTIONS A.K.A. SUB-ROUTINES

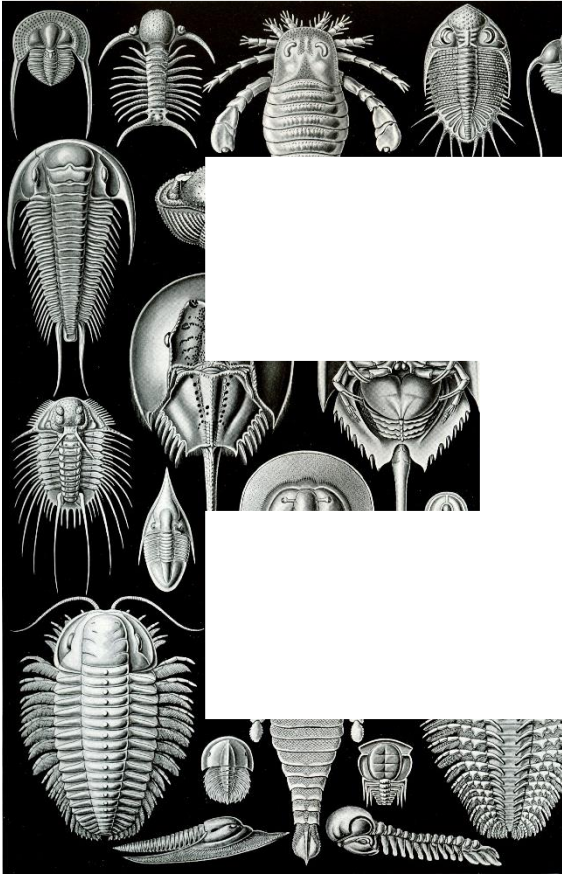
- Let's draw!
- One function:
 1. Iterates
 2. Operates

Assumes `putpixel()`

1. Available;
2. Correct signature;
3. Does the right thing;
4. Returns control to caller!

```
void drawline(int x0, int y0, int x1, int y1) // Partial Bresenham
{
    int dy=y1-y0;
    int x=x0;
    int y=y0;
    int p=2*dy-dx;
    while(x<x1) // 1. iterate
    {
        putpixel(x,y,7); // 2. do something: call putpixel()
        if(p>=0)
        {
            y=y+1;
            p=p+2*dy-2*dx;
        }
        else
        {
            p=p+2*dy;
        }
        x=x+1;
    }
}
```

“

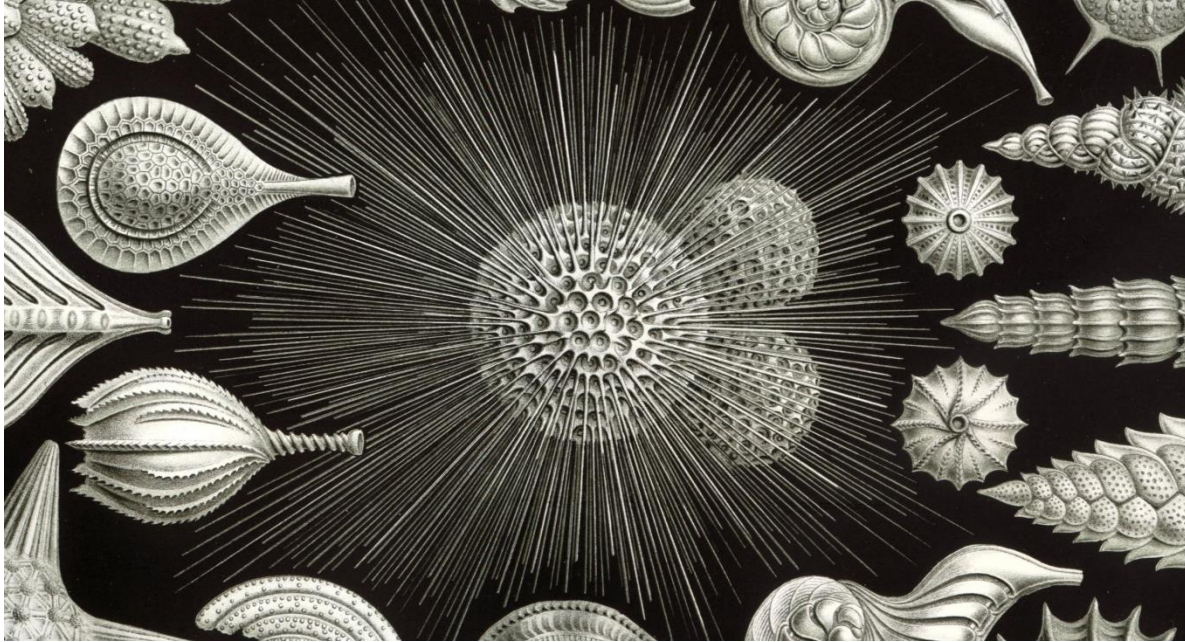


SUBROUTINES ARE EAGER AND CLOSED

EAGER PROCESSING

“Closed” in the sense that they only return after they have iterated over the whole sequence. They eagerly process a whole sequence.

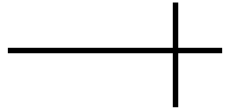
CALLBACKS



EXTERNAL CALLABLES

- Function pointers
- Lambdas
- Callable template parameters or Concepts

- **INVERSION-OF-CONTROL**
- **CALLBACK HELL**
- **STILL EAGER**





CAN WE BREAK THEM OPEN?

If only there was a way to “flip” these iterating functions “inside-out” and iterate over a sequence without pre-committing to a specific operation.

ITERATORS

- Iterator Objects and Iterator Adaptors
 - “Stand-alone” types;
 - Often indirectly or implicitly coupled to a sequence
- Examples from the C++ standard:
 - `std::istream_iterator`
 - `std::reverse_iterator`
 - `std::recursive_directory_iterator`



1998

DID YOU KNOW?

The concept of Iterators has been with C++ since the STL was designed by Alex Stepanov and together with the rest of the STL became part of C++98.

USER DEFINED ITERATORS

OpenCV's `cv::LineIterator`

- Typical Iterator API
- No explicit sequence
- *Lazily generate* elements
- Incremental access to pixels along a line

```
class LineIterator
{
public:
    // creates iterator for the line connecting pt1 and pt2 in img
    // the 8-connected or 4-connected line will be clipped on the image boundaries
    LineIterator( const Mat& img, Point pt1, Point pt2, int connectivity = 8);
    uchar* operator *();           // returns pointer to the current pixel
    LineIterator& operator ++();    // prefix increment operator (++it). shifts
    iterator to the next pixel

    // public (!!!) members [ <groan 🙄> ]
    uchar* ptr;
    const uchar* ptr0;
    int step, elemSize;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;
};
```



USER DEFINED ITERATORS

Example Usage

```
cv::LineIterator it(img, pt1, pt2, 8);
std::vector<cv::Vec3b> buf(it.count);
for(int i = 0; i < it.count; ++i, ++it) // copy pixel values along the line into buf
    buf[i] = *(const cv::Vec3b*)it;
```

```
class LineIterator
{
public:
    // creates iterator for the line connecting pt1 and pt2 in img
    // the 8-connected or 4-connected line will be clipped on the image boundaries
    LineIterator( const Mat& img, Point pt1, Point pt2, int connectivity = 8);
    uchar* operator *();           // returns pointer to the current pixel
    LineIterator& operator ++();    // prefix increment operator (++it). shifts
    iterator to the next pixel

    // public (!!!) members [ <groan 😞> ]
    uchar* ptr;
    const uchar* ptr0;
    int step, elemSize;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;
};
```



OBJECTS THAT LAZILY GENERATE VALUES ARE CALLED
GENERATORS

AWKWARD COUPLING

When do we stop incrementing?

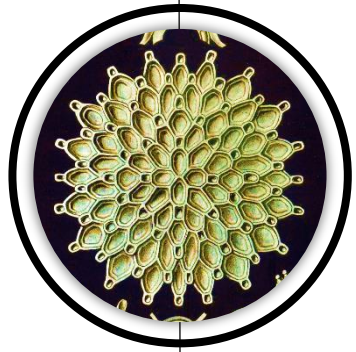
- `cv::LineIterator`: at most `it.count` times
- `std::istream_iterator`: when `== std::istream_iterator()`
- `std::reverse_iterator`: when `== sequence rend()`
- `std::recursive_directory_iterator` when `== std::end(it)`

PITFALL! USER SIDE RUNTIME COUPLING OF BEGIN AND END



RANGES

- Abstraction layer on top of iterators
- The answer to **The Awkward Coupling**
- C++20 Ranges encapsulate:
 - A begin and end iterator-pair;
 - An iterator + size;
 - An iterator and stopping condition
- A single object makes STL algorithms more powerful by making them composable.
- Create pipelines to transform values



2020.

DID YOU KNOW?

Ranges are coming to C++20 and are an amazing new addition to the standard library! Three pillars: Views, Actions, and Algorithms.

DISTRIBUTED LOGIC

Cousin of **Callback Hell**:

- Distributed logic:
 - Logic split between ctor and methods like operator++
- Centralized-state:
 - Intermediate computation variables stored as (mutable) members.

```
class LineIterator
{
public:
    // creates iterator for the line connecting pt1 and pt2 in img
    // the 8-connected or 4-connected line will be clipped on the image boundaries
    LineIterator( const Mat& img, Point pt1, Point pt2, int connectivity = 8);
    uchar* operator *();           // returns pointer to the current pixel
    LineIterator& operator ++();    // prefix increment operator (++it). shifts
                                   iterator to the next pixel

    uchar* ptr;
    const uchar* ptr0;
    int step, elemSize;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;
};
```

```
//...
inline uchar* LineIterator::operator *()           // trivial
{ return ptr; }

inline LineIterator& LineIterator::operator ++() // loop iteration logic
{
    int mask = err < 0 ? -1 : 0;
    err += minusDelta + (plusDelta & mask);
    ptr += minusStep + (plusStep & mask);
    return *this;
}
//...
```

LOGIC PUZZLE

```
void processLine(const Mat& img, Point pt1, Point pt2,...)
{
    // local variables (cv::LineIterator member variables)
    uchar* ptr;
    const uchar* ptr0;
    int step, elemSize;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;

    // initialize local variable (cv::LineIterator::LineIterator() ctor)
    // ...

    // Now draw the line
    for(int i = 0; i < count; ++i) // the explicit loop
    {
        // calculate the next element (LineIterator::operator++())
        int mask = err < 0 ? -1 : 0;
        err += minusDelta + (plusDelta & mask);
        ptr += minusStep + (plusStep & mask);

        doSomething(ptr); // <<!!! ptr is the "current" element/pixel
    }
}
```

**CENTRALIZED LOGIC
BUT: EAGER & CLOSED**

```
class LineIterator
{
public:
    // creates iterator for the line connecting pt1 and pt2 in img
    // the 8-connected or 4-connected line will be clipped on the image boundaries
    LineIterator( const Mat& img, Point pt1, Point pt2, int connectivity = 8);
    uchar* operator *(); // returns pointer to the current pixel
    LineIterator& operator ++(); // prefix increment operator (++it). shifts
    iterator to the next pixel
};
```

```
uchar* ptr;
const uchar* ptr0;
int step, elemSize;
int err, count;
int minusDelta, plusDelta;
int minusStep, plusStep;
```

**DISTRIBUTED LOGIC
YET: LAZY & OPEN**

```
//...
inline uchar* LineIterator::operator *() // trivial
{
    return ptr;
}

inline LineIterator& LineIterator::operator ++() // loop iteration logic
{
    int mask = err < 0 ? -1 : 0;
    err += minusDelta + (plusDelta & mask);
    ptr += minusStep + (plusStep & mask);
    return *this;
}
//...
```



CAN WE HAVE NICE THINGS?

If only there was a way to write easy to reason about, serial algorithms with local scoped variables while still abstracting away the iteration...

COROUTINES

“Coroutines make it trivial to define your own ranges.”
— Eric Niebler



A Coroutine is a **function** that:

1. Can suspend execution;
2. Return an intermediate value;
3. Resume later;
4. Preserve local state;
5. Allows re-entry more than once;
6. Non-pre-emptive → Cooperative

JUST LIKE WHAT WE WANT!

1958 •

DID YOU KNOW?

The term *coroutine* was coined by Melvin Conway in 1958. Boost has had several coroutine libraries at least since 2009 and some C coroutine libraries were well known since before 2000.

COROUTINES

"Coroutines make it trivial to define your own ranges."
— Eric Niebler



A Coroutine is a **function** that:

1. Can suspend execution;
2. Return an intermediate value;
3. Resume later;
4. Preserve local state;
5. Allows re-entry more than once;
6. Non-pre-emptive → Cooperative

JUST LIKE WHAT WE WANT!

```
void processLine(const Mat& img, Point pt1, Point pt2,...)
{
    // local variables (cv::LineIterator member variables)
    uchar* ptr;
    const uchar* ptr0;
    int step, elemSize;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;

    // initialize local variable (cv::LineIterator::LineIterator() ctor)
    // ...

    // Now draw the line
    for(int i = 0; i < count; ++i) // the explicit loop
    {
        // calculate the next element (LineIterator::operator++())
        int mask = err < 0 ? -1 : 0;
        err += minusDelta + (plusDelta & mask);
        ptr += minusStep + (plusStep & mask);

        ? doSomething(ptr); // <<!!! ptr is the "current" element/pixel
    }
}
```

C++20 COROUTINES

- The answer to **Distributed Logic**
- A **function** is a coroutine if any of the following:
 - Uses `co_await` to suspend execution until resumed;
 - Uses `co_yield` to suspend + returning a value;
 - Uses `co_return` to complete + return a value.
- Return type must satisfy certain requirements.

CANNOT TELL COROUTINE FROM FUNCTION BY SIGNATURE

COROUTINES ARE AN “IMPLEMENTATION DETAIL”



2020 •

DID YOU KNOW?

The data required to resume coroutine execution is stored separately from the caller-stack. To make this even more confusing they are called *Stackless* do distinguish them from *Stackful* coroutines which use CPU/OS fibers)).

THE SIMPLEST CODE

```
auto zoro() { return 42; }
```

- What does `zoro()` return? **42**
- The return type is... **int**
- Is it a coroutine? **No**



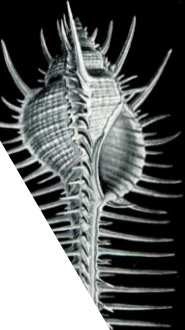
THE SIMPLEST CODE

```
auto zoro() { return 42; }
```

- What does `zoro()` return? **42**
- The return type is... **int**
- Is it a coroutine? **No**

```
auto coro() { co_yield 42; }
```

- What does `coro()` return? **Not 42**
- The return type is... ? **Not int**
- Is it a coroutine? **Yes**

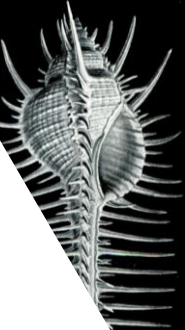


THE SIMPLEST CODE

```
for (auto v: coro())  
    cout << v;
```

OR

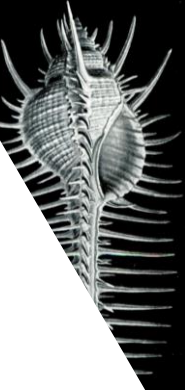
```
auto gen = coro();           // the (suspended) generator  
auto it = gen.begin();       // the iterator: resumes the coroutine, executing it until it  
                              // encounters co_yield  
cout << *it;                 // dereference to get the actual value.  
  
// or alternatively  
cout << *coro().begin();
```



INFINITE RANGES

```
auto iota(unsigned int n = 0)
{
    while(true)
        co_yield n++;
}

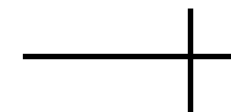
// usage:
std::copy_n(iota(42).begin(), 9, std::ostream_iterator<int>(std::cout, ", "));
// prints: 42,43,44,45,46,47,48,49,50
```



DECEPTION



- NO AUTO RETURN TYPE
- NO STD CORO LIBRARY!



MSVC EXTENSIONS

- Non-conforming MSVC infers `std::experimental::generator<T>` for `auto`
- No such thing as `std::experimental::generator<T>`
- Until then, use e.g. Lewis Baker's `cppcoro`

```
auto spiral()
{
    int x = 0, y = 0;
    while (true)
    {
        co_yield Point{ x, y }; // yield the current position on the spiral
        if (abs(x) <= abs(y) && (x != y || x >= 0))
            x += ((y >= 0) ? 1 : -1);
        else
            y += ((x >= 0) ? -1 : 1);
    }
}
```

SPIN CYCLE

```
auto hueCycleGen(int step = 1)
{
    Mat3b rgb(1,1), hsv(1,1);
    hsv(0,0) = { 0, 255, 255 }; // { Hue=0, Full Saturation, Full Intensity }
    while (true)
    {
        cvtColor(hsv, rgb, COLOR_HSV2RGB_FULL);
        co_yield rgb(0,0); // yield the current RGB corresponding to the current HSV.
        (hsv(0,0)[0] += step) %= 255; // cycle the H channel
    }
}
```




```
template <typename T, typename U>
auto zip(T vals1, U vals2)
{
    auto it1 = vals1.begin();
    auto it2 = vals2.begin();
    for (; vals1.end() != it1 && vals2.end() != it2; ++it1, ++it2)
        co_yield std::make_pair(*it1, *it2);
};
```

SPIN CYCLE

```
for (auto [pos, color] : zip(spiral(), hueCycleGen())) // 1. zip the generators
{
    cv::Point pix = pos + offset;                      // 2. offset to actual pixel
    position                                           // 2. offset to actual pixel
    if (img.rows*2 <= pix.x && img.cols*2 <= pix.y)    // 3. no more pixels to scan
        break;
    if (!rect.contains(pix))                          // 4. skip out of bounds
        continue;
    img(pix) = color;                                 // 5. set pixel color
}
```



```

class TreeNode
{
    // ...
    using ValueGen = std::experimental::generator<int>;
    ValueGen inorder() // In-order (Left, Root, Right)
    {
        if (left_)
            for (auto v : left_->inorder()) // iterate on recursion
                co_yield v;
        co_yield val_;
        if (right_)
            for (auto v : right_->inorder())
                co_yield v;
    }

    ValueGen preorder() // Pre-order (Root, Left, Right)
    {
        co_yield val_;
        if (left_)
            for (auto v : left_->preorder())
                co_yield v;
        if (right_)
            for (auto v : right_->preorder())
                co_yield v;
    }

    ValueGen postorder() // Post-order (Left, Right, Root)
    { /* ... */ }
}

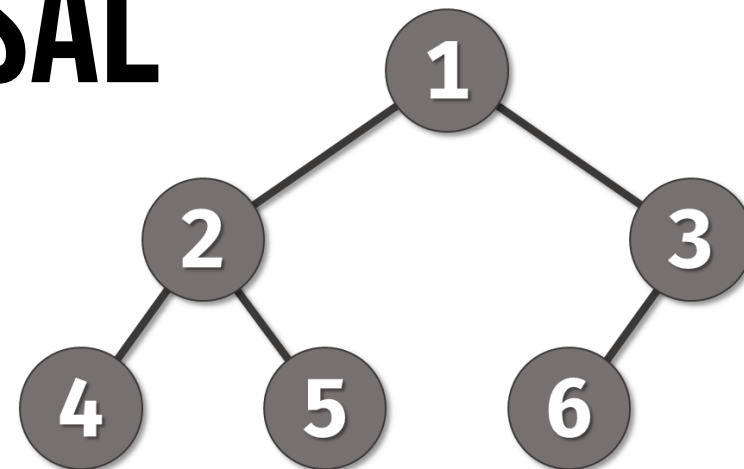
```

```

enum Order { IN_ORDER, PRE_ORDER, POST_ORDER };
auto order(Order order) // this is NOT a coroutine!
{
    switch (order)
    {
        case IN_ORDER: return inorder();
        case PRE_ORDER: return preorder();
        case POST_ORDER: return postorder();
    }
}

```

TREEVERSAL



```

for (auto val : head.order(TreeNode::IN_ORDER))
    std::cout << val << ", "; // 4, 2, 5, 1, 6, 3

```

```

class TreeNode
{
    // ...
    using ValueGen = std::experimental::coroutine::generator<int>;
    ValueGen inorder() // In-order traversal
    {
        if (left_)
            for (auto v : left_->inorder())
                co_yield v;
        co_yield val_;
        if (right_)
            for (auto v : right_->inorder())
                co_yield v;
    }

    ValueGen preorder() // Pre-order (Root, Left, Right)
    {
        co_yield val_;
        if (left_)
            for (auto v : left_->preorder())
                co_yield v;
        if (right_)
            for (auto v : right_->preorder())
                co_yield v;
    }

    ValueGen postorder() // Post-order (Left, Right, Root)
    { /* ... */ }
}

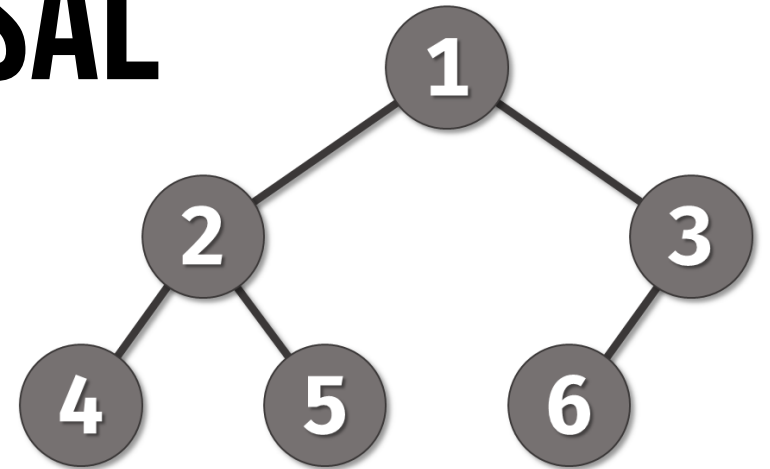
```

```

cppcoro::recursive_generator<int> inorder() // Inorder (Left, Root, Right)
{
    if (left_) co_yield left_->inorder();
    co_yield val_;
    if (right_) co_yield right_->inorder();
}

```

TREEVERSAL

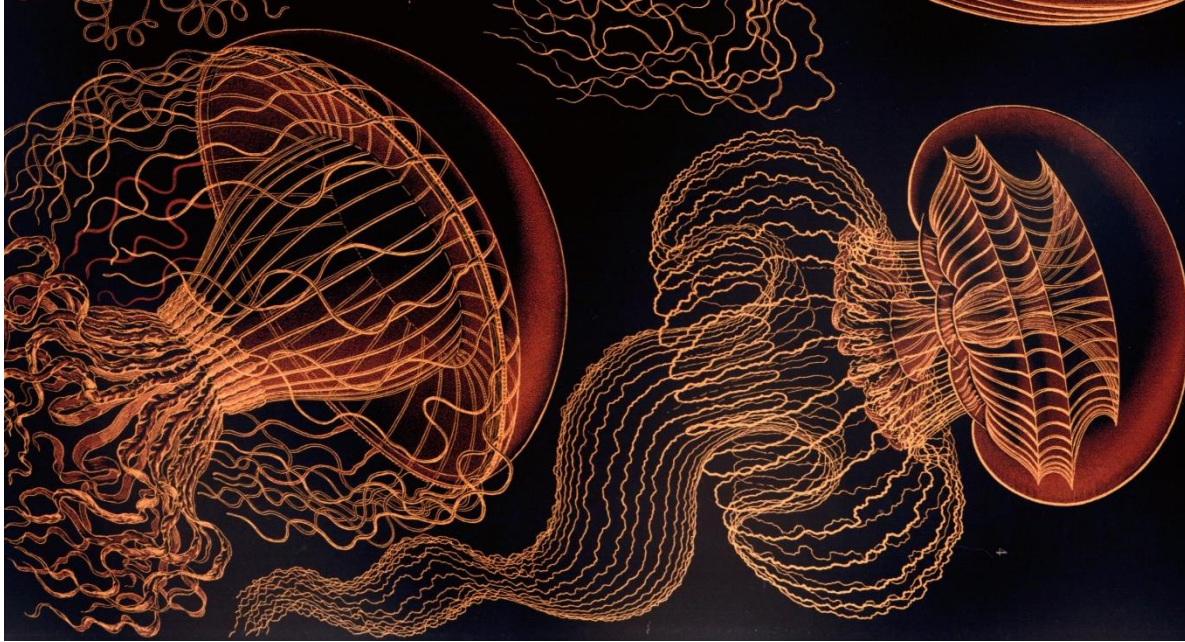


```

for (auto val : head.order(TreeNode::IN_ORDER))
    std::cout << val << ", "; // 4, 2, 5, 1, 6, 3

```

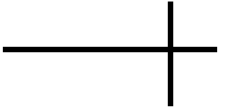
PITFALLS



NOT PERFECT YET

- Beware of temporaries and references
- Pass by value
- Beware of inadvertent execution

- DANGLING REFERENCES
- LIMITATIONS



DANGLING REFERENCES

- Coroutine execution starts *after* calling `begin()`
- `s` is a ref to temp string which goes out of scope before it is executed!

BOOM!

```
generator<char> explode(const std::string& s)
{
    for (char ch : s)
        co_yield ch;
}

int main()
{
    for (char ch : explode("hello world"))
        std::cout << ch << '\n';
}
```

From blog post by Arthur O'Dwyer bit.ly/2NDSF9G

TIP: TAKE COROUTINE ARGUMENTS BY VALUE

DANGLING REFERENCES

- C
- Ca
- S
- Ol

```
{  
    init-statement  
    auto && __range = range_expression ;  
    auto __begin = begin_expr ;  
    auto __end = end_expr ;  
    for ( ; __begin != __end; ++__begin) {  
        range_declaration = *__begin;  
        loop_statement  
    }  
}
```

(since C++20)

Temporary range expression

If *range_expression* returns a temporary, its lifetime is extended until the end of the loop, as indicated by binding to the forwarding reference `__range`, but beware that the lifetime of any temporary within *range_expression* is not extended.

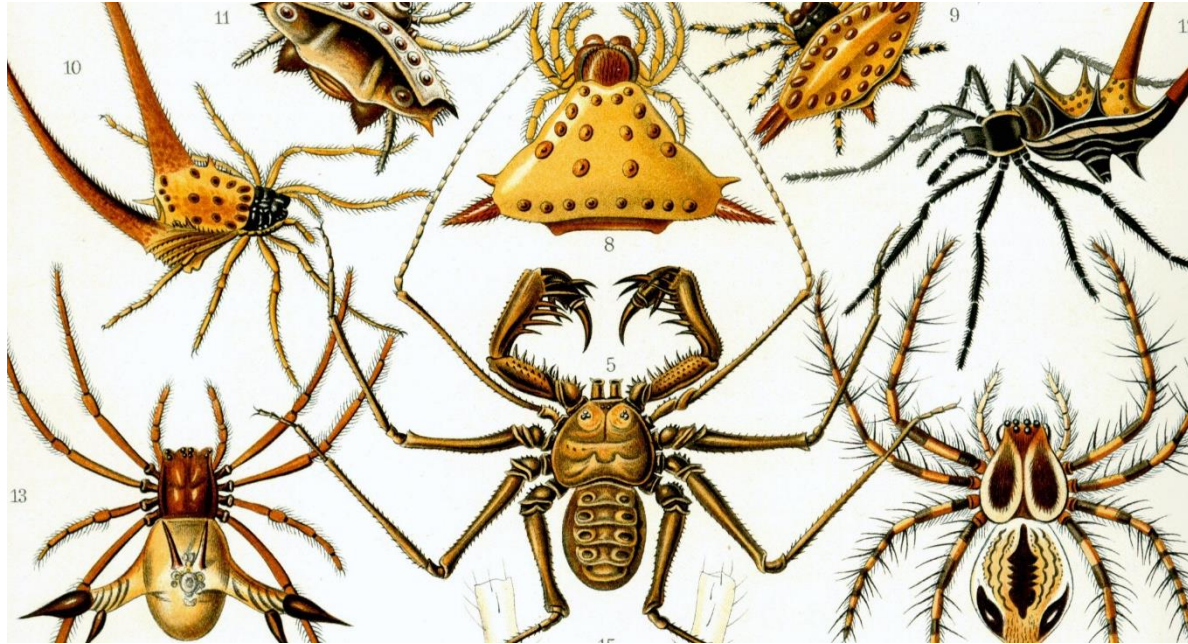
```
for (auto& x : foo().items()) { /* .. */ } // undefined behavior if foo() returns by value
```

This problem may be worked around using *init-statement* (C++20):

```
for (T thing = foo(); auto& x : thing.items()) { /* ... */ } // OK
```

(since C++20)

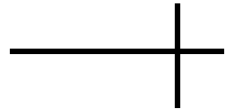
LIMITATIONS



NOT PERFECT YET

- No plain return statements
- No placeholder return types (auto or Concept)
- `constexpr` functions, constructors, destructors, and the main function cannot be coroutines

- MISSING FEATURES
- NO STD CORO LIBRARY!
- QOI LIBRARY ISSUES
- QOI COMPILER ISSUES



RESOURCES



- A massive list of coroutine resources, MattPD bit.ly/3436zZ3
- en.cppreference.com/w/cpp/language/coroutines
- The `#coroutines` channel on the C++ Slack
- More details on my blog videocortex.io/2019/Brain-Unrolling

THANK YOU!

@adishavit :: videocortex.io