Coroutine Intuition

Roi Barkan

C++ now 2023

Link to Slides

Hi, I’m Roi

- Roi Barkan (he/him) - רועי ברקן
- I live in Tel Aviv
- C++ developer since 2000
- SVP Development & Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- Always happy to learn and explore
  - Please - ask questions, make comments

roi@istraresearch.com

Outline

- Introdution - Intuition
- What are Coroutines
- Stacks, bytes, details
- Some mechanics
- Coroutine Scenarios
  - Generators
  - Protocol state machines
  - Cooperative asynchrony
  - Schedulers
  - Thread hopping
  - Senders, receivers
- Summary

roi@istraresearch.com

Intuition - When is a Language Feature Useful ?

- C++ supports many paradigms, architectures and idioms
  - Paradigms: Object Oriented, Functional, Data Oriented, Generic, etc.
  - Architectures: Pub-sub, Client-server, Concurrency & Parallelism, Data-flow streams, etc.
  - Idioms/techniques: RAII, Range-oriented, Virtual-method polymorphism, Type-erasure, etc.
- Various language features lend themselves to specific uses and designs
  - Concepts aim for compile-time dispatch, shared_ptrs help with OOP object graphs, range-adapters ease stream oriented approaches, etc.
- Which system designs can benefit most from coroutines ?
  - Are coroutines considered 'implementation details' ?
  - How 'viral' are coroutine approaches in a code-base ?
  - When will coroutines make our systems: simpler ? faster ? easier to maintain ?

roi@istraresearch.com

Analogies of Feature Usefulness

- STL Containers
  - High performance data structures
  - Easy to use, complicated to implement, heavy usage of templates
  - Work great for large collections of ‘algebraic value types’
- Lambdas
  - ‘Syntactic sugar’ of C++98 function-objects
  - Simplifies our code through shorter, more localized code - the compiler will figure out the rest
- Ranges
  - Simple notion with very complex machinery under the hood
  - Enlarges our vocabulary: adapters, views, sentinels, composable algorithms
  - Most suited for data-oriented one dimensional systems/processors

roi@istraresearch.com

Coroutine Intuition

- Coroutines add new words to our *design vocabulary*
  - Generators, schedulers, tasks, continuations, …
    - Conceptually, those existed before coroutines, but now can be more common/idiomatic
- Coroutines can simplify or canonize otherwise complex component interactions
  - Asynchronous systems, complex state machines, massive concurrency
- Coroutines support different code organization
  - Differentiate between code-locality and execution locality
  - Separate algorithm logic and its execution policy
- I’ll focus more on how/when we write coroutines and less on the generated binary
  - Separate the concerns of library authors and application developers

roi@istraresearch.com

What are Coroutines ?

History of Coroutines

- Coroutines - one of C++20 big-4 features
- Technically - functions that can ‘suspend’ and be ‘resumed’
  - suspend might happen on `co_yield` or `co_await`
  - resume by calling `std::coroutine_handle<Promise>::resume()`
- Old idea, seemed abandoned, reborn
  - 1950s- Term coined in 1958
  - 1960s- Simula
  - 1970s- Modula-2
  - 2000s- C# generators, D, python, Kotlin, Scala
  - 2010s- C#, Javascript, Rust, Go
  - 2020s- C++

roi@istraresearch.com

Suspending and Resuming

- In order to resume a function, its state needs to be saved
    - Local variables
    - Arguments
    - “Machine state” - temporaries, registers, exact suspension point
    - Return address? Return value?
- Every function call is a suspend-resume
    - Less general than coroutines -
        - Every suspend jumps to the start of a function
        - Resume when the subroutine returns (and is destroyed)
    - LIFO resume order - ideal for the stack data structure.

roi@istraresearch.com

The Stack, its Frames, Suspend-Resume

- Information used by a function needs to be stored somewhere
  - Arguments, local variables, temporaries, sub-expressions, data (registers) the caller expects to be saved
- Before calling a function, more information should be stored
  - Place for the return value, exact resume point (return address),
    data (registers) the callee expects to freely use
- Coroutines need the same information but suspend-resume should be more flexible

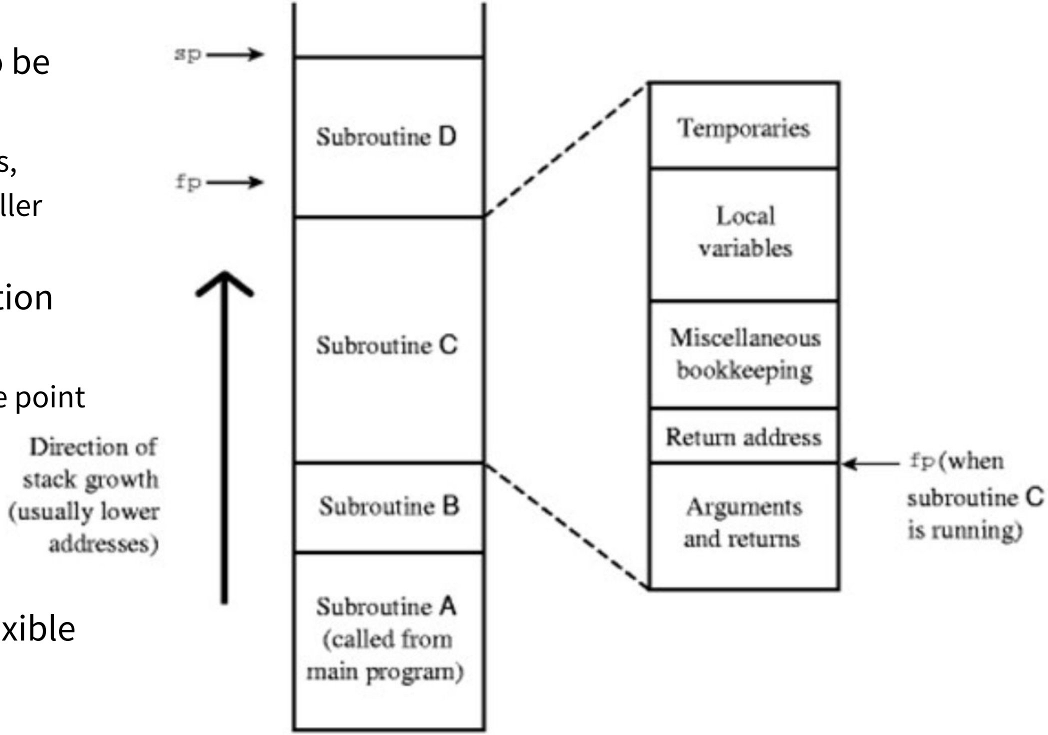


Image Credit

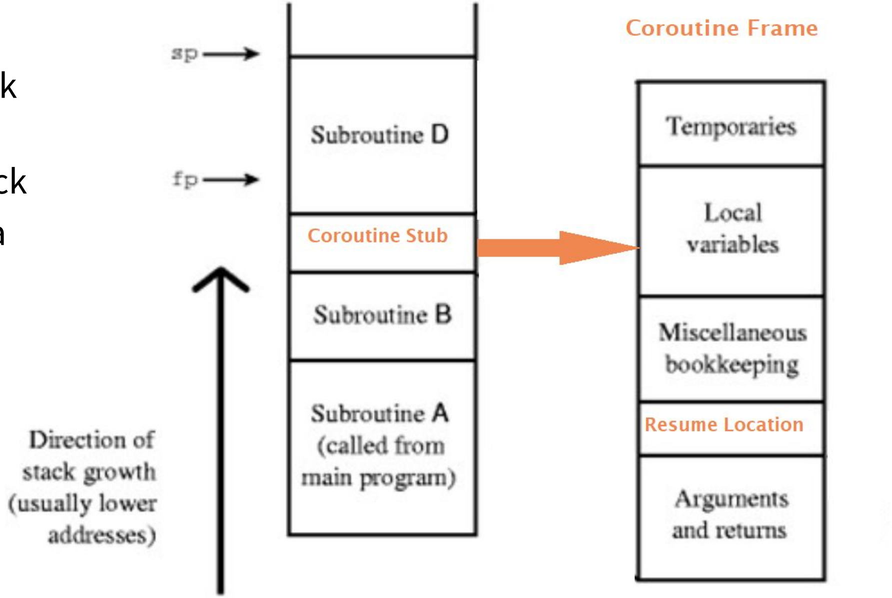
Suspending and Resuming - Coroutines

- Coroutines add flexibility to the control-transfer mechanism
  - Suspend and jump to the middle of another suspended coroutine or my (non coroutine) caller
  - decouple stack-frame creation/destruction and the execution control-flow
- Stackful vs. Stackless
  - Stackful - invocation of a coroutine creates a new separate stack (this is not C++20)
    - High similarity with fibers and/or ‘green threads’
  - Stackless - the “coroutine frame” is created (allocated) away from the stack (operator new)
- How does a stackless coroutine call functions?

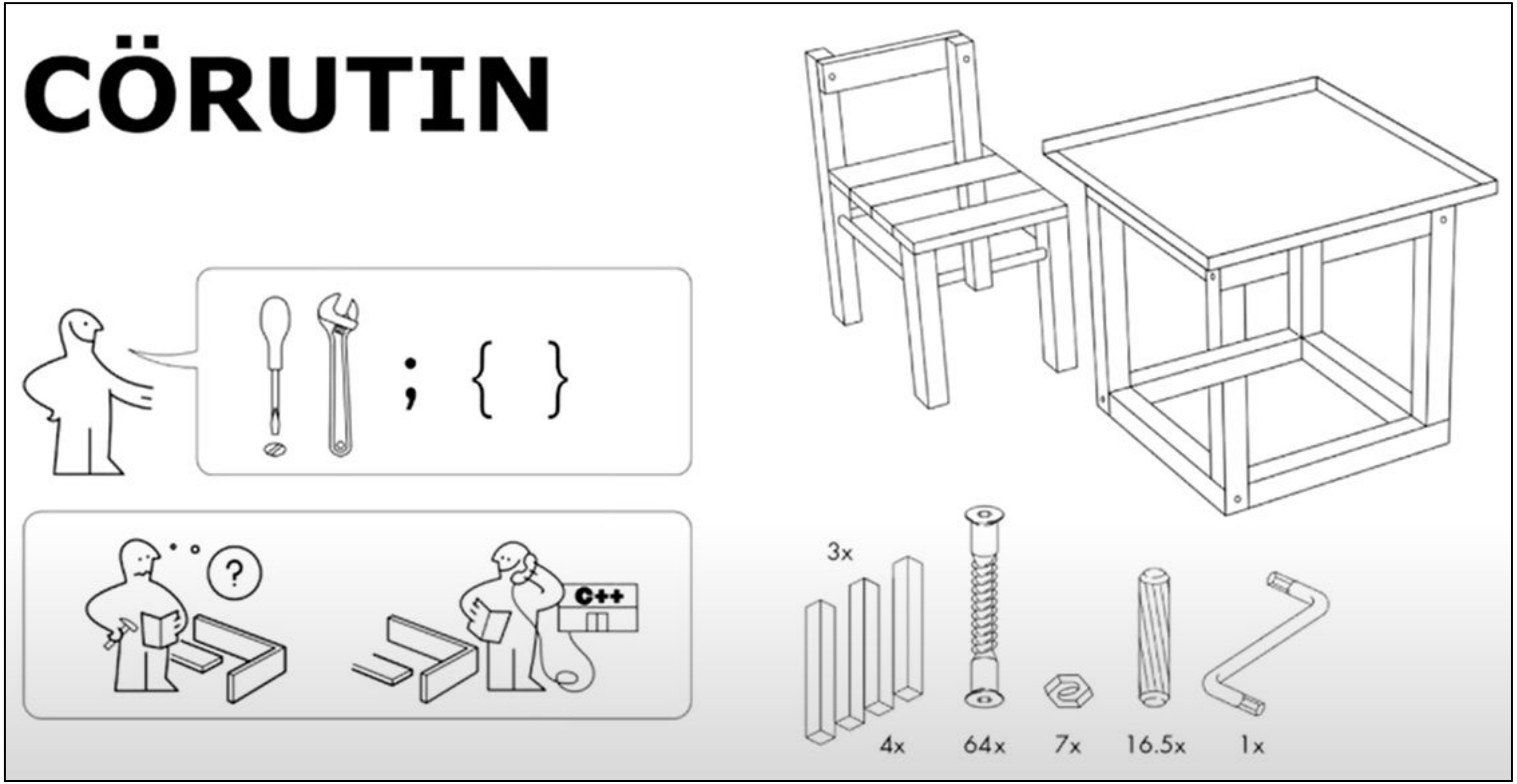
roi@istraresearch.com

Stackless Coroutines

- Coroutine frame is away from the stack
- Resume location also away from the stack
    - Relevant when suspended
- Return address (back from D)- on the stack
- Calling/Resuming isn’t much more than a virtual function call (indirect jump)
- No way to resume Subroutine B before completion of Subroutine D.
    - No way to suspend it either



Mechanics of Coroutines



Kevlin Henney

roi@istraresearch.com

Terminology - the Coroutine Objects

- Coroutine frame - keeps the state of a coroutine
- Promise - controller object of a coroutine
  - Controls whether to suspend and/or resume when coroutine starts/ends/yields
    - implements functions like `initial_suspend()`, `final_suspend()` and `yield_value(value)`
  - Created and destroyed with a coroutine
  - Conceptually - a part of the coroutine frame (or derived from a compiler controlled frame)
- `std::coroutine_handle<Promise>` - pointer/accessor to a frame
  - Used to `resume()` or `destroy()` a frame
  - Bidirectional access of the `Promise`
- Return object - the caller facing API of a coroutine
  - The declared return type of the coroutine.
  - Typically has a `std::coroutine_handle` used to control the coroutine.

roi@istraresearch.com

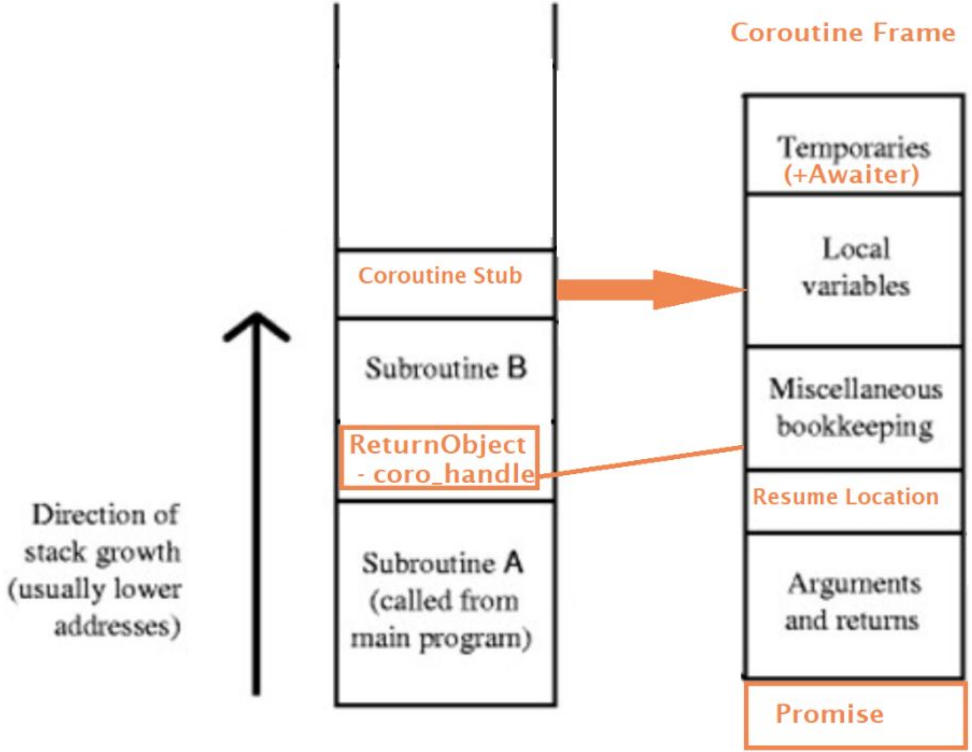
Return Objects - the API of a coroutine

- The basic `resume()` and `destroy()` calls are considered low level
- Coroutines are meant to represent some higher level notion, which internally uses the suspend-resume functionality as ‘implementation details’
- For example - a coroutine that sequentially `co_yield`s values of the same type can respresent a *sequence* or a *range* which can be iterated over
  - `generator<T>` is a return object that has a sequence API.

External Suspend-Resume Control - co_await

- co_await allows a coroutine to give another object (not the promise) control
- Such objects are called **Awaitable**. They can create **Awaiter**s
  - Typically **Awaitable** is an **Awaiter**, or implements **operator co_await** to create one. The promise can also assist
- Once created, an **Awaiter** decides whether to suspend this coroutine, what to resume in case of suspend, and how to eventually resume this coroutine:
  - bool await_ready() - should this coroutine continue now ?
  - void/bool/coroutine_handle<> await_suspend(coroutine_handle<Promise>) - handle the suspend, decide what to do next
  - void/Obj await_resume() - handle the resume, pass a value into the resumed coroutine.

Example Memory Layout - `co_await`



roi@istraresearch.com

Coroutine Scenarios

`generator<T>` - Multiple `co_yield` as a Range

- Suspend as a mechanism to return multiple values
- Upon suspend the caller (typically non coroutine) is resumed, and uses the value
- ‘Lazy sequence’
- Implement `iterator`/`sentinel` helper objects that allow range-like access
- Example:

```
std::generator<uint64_t>
fibonacci_sequence(unsigned n) {
  if (n > 0) co_yield 0;
  if (n > 1) co_yield 1;
  uint64_t a=0, b=1;
  for (unsigned i = 2; i < n; i++) {
    uint64_t s=a+b;
    co_yield s;
    a=b;
    b=s;
  }
}
```

Composing Coroutines - Continuations

- Splitting a coroutine into sub-coroutines isn't trivial
    - Stackless nature means that the typical function-call suspend-resume isn't the default
- Objective - a coroutine can call another, and resume when it completes
    - Caller should create a coroutine-frame, suspend and start it instead of returning to its caller
    - Callee should resume its calling coroutine instead of returning back to the stack
- `task<T>` - a coroutine return-object that is also an `Awaiter`
    - Caller `co_await`'s the `task` which it creates.
    - `task<T>::await_suspend()` stores the caller-handle (in its promise) and starts the callee
    - The stored handle is called the ***continuation***
    - `task<T>::final_suspend()` resumes the continuation
    - Also deals with exceptions
- Essentially a linked-list of coroutine frames forms a substitute stack.

Asynchronous I/O

- Operating system objects (sockets, files) provide **`Awaitable`**s
- Example:

```
char data[1024];
while (true) {
  size_t n = co_await inSocket.async_read_some(buffer(data));
  co_await async_write(outSocket, buffer(data, n));
}
```

- `await_suspend()` stores the `coroutine_handle<>`, and registers a callback with the O/S
- The callback will put results in the **`Awaiter`** and `resume()` the coroutine.
  - Typically some **`io_service`** object will have the functionality of polling events and executing the callbacks
- `await_resume()` provides the results to the coroutine.

Massive Concurrency

- Concurrent execution of long-running actors/algorithms
  - Example: AI Characters (NPCs) inside games/simulations
  - Example: maintaining a complex network protocol with many clients
- Utilize the low overhead and fast suspend/resume speed
  - Compared to threads, and even ‘fibers’ (and stackful coroutines)
- ***Scheduler*** object acts as an **`Awaitable`** and all actors **`co_await`** it regularly
  - Alternatively, actors can **`co_yield`** and let their promise pass the handle to the scheduler
  - Some physical thread(s) give the ***scheduler*** an opportunity (endless loop?) to run the actors
- Known as ***Cooperative Multitasking***
  - Actors are responsible not to hog their system
  - Simpler code with less concern about mutexes and races.

Reordering / Prioritizing Work

- Enhance the notion of the scheduler for smarter sequencing of work
- Priority - once a task completes urgent work, it can `co_yield` to allow other tasks to take place without losing context
- Batching - the scheduler can look across suspended coroutines and interact with **`Awaiters`** to generate their desired data/results efficiently
- Async versions of mutexes/barriers to avoid blocking worker threads waiting for synchronization.

Switching Execution Environment

- Once a coroutine is suspended, it can be resumed by another thread or in another environment
- Example: switch to/from the UI thread
- Example: switch to/from GPU threads (inspirational)
- Example: Split/shard data ownership between threads to avoid the need for data synchronization mechanisms (mutexes, atomics, etc.).

Task Graphs - Senders and Receivers

- WG21 vision for concurrent algorithms entails the creation of linked tasks passing data through them as a composable mechanism for large scale systems and algorithms
    - Senders - components that can generate sequences of data
    - Receivers - components that get notified when data arrives
    - Task graphs are fully created, giving ‘executors’ a chance to optimize them before starting the data flow
- Observation: **`Awaitable`** types are Senders, Senders are **`Awaitable`**
- Coroutine’s light-weight and statically-types nature allow potential optimization of task graphs.

roi@istraresearch.com

Senders vs. Awaiters

```cpp
// This is a coroutine:
unifex::task<int> read_socket_async(socket, span<char, 1024>);


unifex::task<void> concurrent_read_async(socket s1, socket s2)
{
  char buff1[1024];
  char buff2[1024];

  auto [cbytes1, cbytes2] =
      co_await std::execution::when_all(
          read_socket_async(s1, buff1),
          read_socket_async(s2, buff2)
      ) | into_tuple();

  /*...*/
}
```

[Niebler CppCon 2021]()

roi@istraresearch.com

Coroutine Anti-Patterns

- Just because something can be done with coroutines doesn’t mean that it should.
- Main reasons:
  - Code vs. execution locality separation can be hard to grasp
  - Performance can rely on non-mandated compiler knowledge/optimization
  - Tricky interoperation with allocation, concurrency and error-handling sub-systems
  - The ‘viral’ natures of continuation chains
- Common pitfall - forgetting there’s another more direct approach
  - Text book state machines
  - Using `co_yield` and `co_await` as tricks to ‘return’ different object types
- Reinventing the wheel
  - Schedulers are complicated
  - Libraries for new language features are relatively infant, keep on the lookout

roi@istraresearch.com

Examples of Anti Patterns

- An event dispatch loop:
  - ```
    while (true) {
      auto event = co_await world;
      switch(event) {
        //… non-coroutine code here
      }
    }
    ```
  - There are so many variations of such loops without coroutines.
  - The uniqueness of coroutines are the multiple suspension points and lifetime of local variables

roi@istraresearch.com
    ```

Examples of Anti Patterns (2)

- State preserving function split:
    - ```
      std::generator<StageEnum> multi_stage_work() {
        //... first stage
        co_yield StageEnum::FIRST;
        //... first stage (may use variables assigned earlier)
        co_yield StageEnum::SECOND;
        // ... and more
      }
      ```
    - An easy (too easy) way to refactor monolith algorithm.
    - Object lifetime management is effectively gone
      ```

Summary - (More Intuition?)

Coroutines are Complicated

- New types of objects, interfaces
- Non-intuitive and potentially complex control flow
- Tons of auto-generated hidden code, yet still tons of boilerplate
- New concerns about object lifetime, cost-of-abstraction, quality of the optimizer
- It all look so much simpler in other languages
- Analogies:
  - C++11 lambdas and C++14 generic lambdas
  - C++11 concurrency model.

roi@istraresearch.com

Coroutines are Powerful

- Potential for enriching our design language with new vocabulary
    - Generators, Schedulers, Tasks, Continuations
- Unmatched performance when done right
    - Compiler is aware of the objects, can inline core and elide allocations
- Gateway towards simpler and safer concurrency
- Analogies:
    - STL containers/algos so powerful even for non template-experts
    - tuple/variant/ranges so commonly used yet difficult to implement.

roi@istraresearch.com

Hard Parts are (Mostly) in Libraries

- Most coroutines can adhere to one of the common paradigms
- Return-Objects, Promises Awaiters for those paradigms are only written once
- Already open-source, soon to be standardized.
  - cppcoro - github.com/lewissbaker/cppcoro
  - unifex - github.com/facebookexperimental/libunifex
  - asio - think-async.com
  - corobatch - github.com/MakersF/corobatch
  - ...
- All that's left is to choose your paradigm and implement your algorithm.

roi@istraresearch.com

Summary

- Coroutines are powerful
- Coroutines are complicated
- Hard part is (mostly) in Libraries

- Thank you !!
    - Questions and comments are welcome

roi@istraresearch.com

Istra Research