Alain Carlucci
alain@rev.ng

Pietro Fezzardi
pietro@rev.ng

September 03 – NDC TechTown 2020

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

revng

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

revng

Alain Carlucci

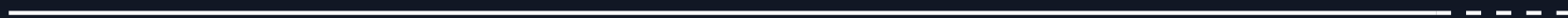
- Software Engineer @ rev.ng
- Working on GUI and tools

Pietro Fezzardi

- CTO @ rev.ng
- Working on decompiler

- We build analysis tools for binary programs
- We are building a decompiler based on LLVM and QEMU
- We do consultancy on compilers and emulators

Frontend
Main Event Loop



Backend
Worker Thread

revng

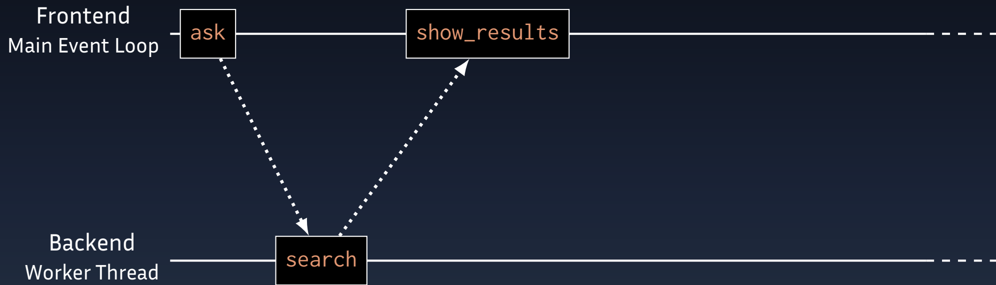
Frontend
Main Event Loop

ask

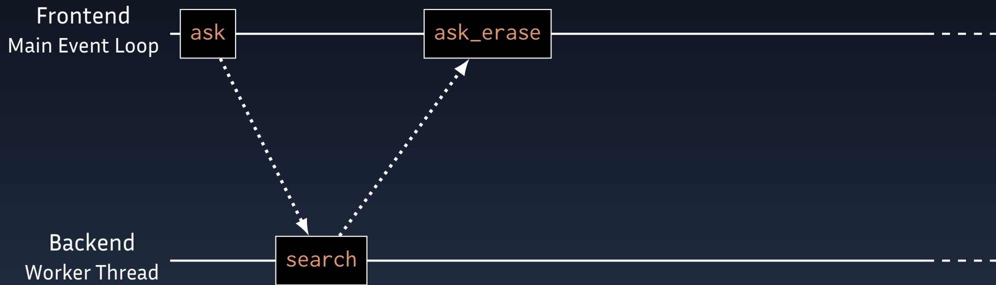
Backend
Worker Thread

revng

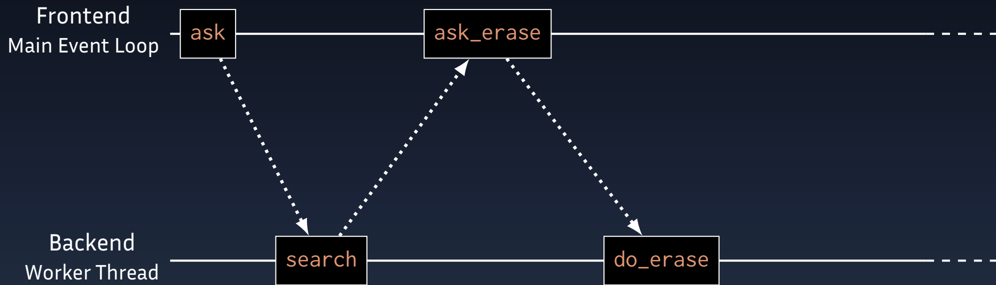
rev.ng



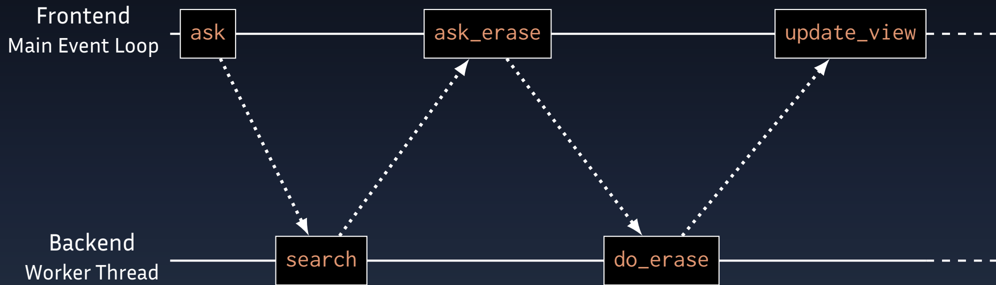
revng



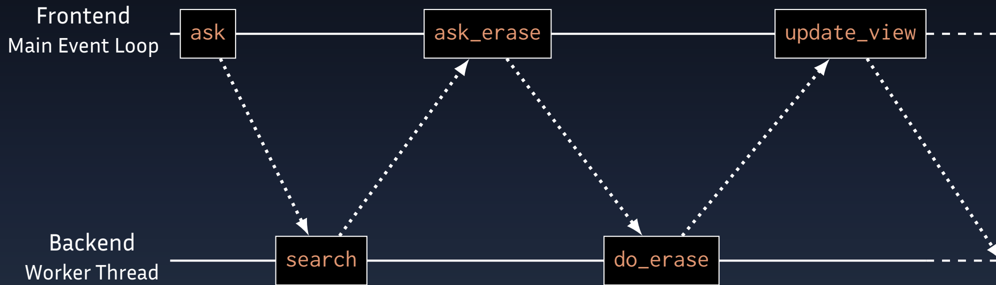
rev.ng



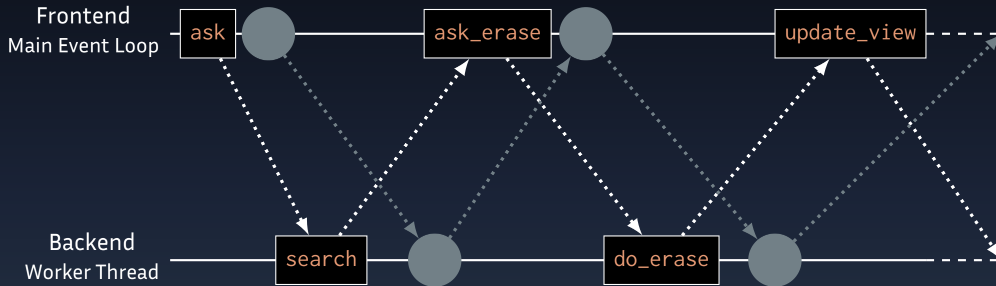
reνng



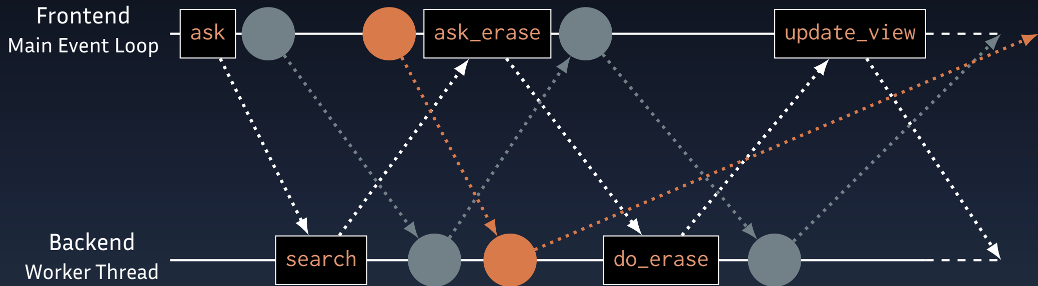
revng



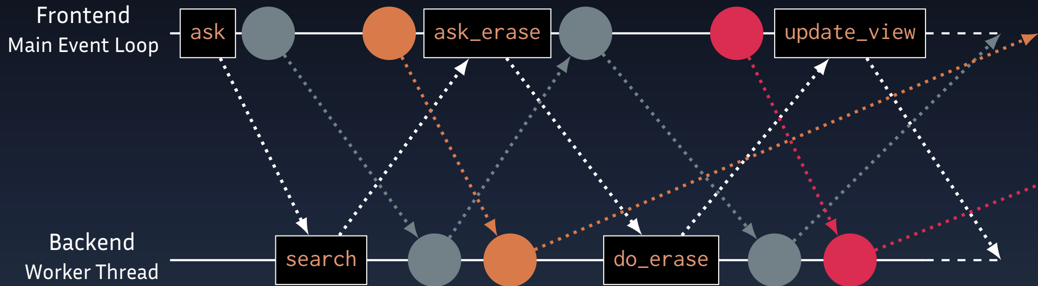
rev.ng



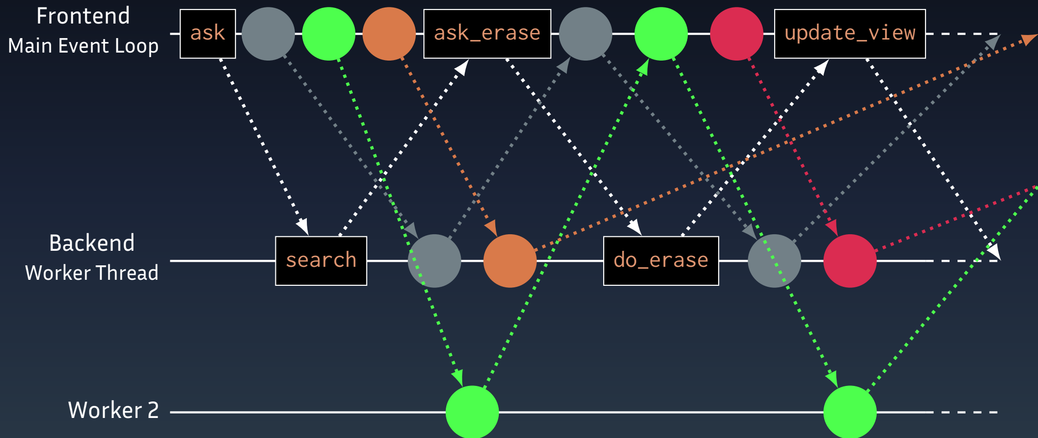
rev.ng



rev.ng



rev.ng


Frontend
Main Event Loop
ask
ask_erase
update_view
Backend
Worker Thread
search
do_erase
Worker 2

- **Asynchronous execution of sequential code** – serial tasks switch threads
- **Clarity** – async language constructs tend to obscure the serial logic
- **Data races** – if worker thread accesses objects managed by main thread
- **Lifetime checks** – if the worker sends results to a window that might be closed
- **Portability** – solve all the previous in a portable way, ideally standard C++

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

rev.ng

- – User chooses to delete a `Thing`
- PING – GUI asks to the backend for the name of the `Thing`
- PONG – Backend searches the `Thing` and, if it exists, sends the name to GUI
- – GUI shows a popup: "Do you want to delete $Name?"; the user clicks YES
- PING – GUI asks the backend to delete the `Thing`
- PONG – Backend deletes the `Thing` and asks the UI to refresh
- – GUI refreshes the screen.

- Backend and Frontend objects are not necessarily thread-safe
- GUI objects should be used only in the main thread
- Backend objects should be used only in the worker thread

rev.ng

- Any Qt thing (widget, class, ...) inherits from the class `QObject`
- Pointers to `QObject` can be wrapped in a `QPointer<T>`
- `QPointer`s are automatically notified when the pointee is deallocated
- On notification of deallocation, `QPointer`s becomes `nullptr`

rev.ng

- Qt uses a notification mechanism built on *signals* and *slots*
- Signals are notifications, Slots are handlers
- A Signal can be connected to multiple Slots, i.e. functions
- If a `QObject` is destroyed, all its connections are destroyed
- This mechanism allows `QPointers` to be automatically nullified

```
void MyWidget::deleteThing(int thingID) {
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty())
      return; // Thing not found
    emit backend->gotNameForThing(result[0]->getName()); // PONG
  });
}
```


rev.ng

```
void MyWidget::deleteThing(int thingID) {
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty())
      return; // Thing not found
    emit backend->gotNameForThing(result[0]->getName()); // PONG
  });
}
```

```
MyWidget::MyWidget() : backend(getBackend()) {
  connect(backend, &BackendObj::gotNameForThing, this, &MyWidget::askDelete);

}
```


rev.ng

```cpp
void MyWidget::deleteThing(int thingID) {
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty())
      return; // Thing not found
    emit backend->gotNameForThing(result[0]->getName()); // PONG
  });
}

void MyWidget::askDelete(QString name) {
  if(!askQuestion("Do you want to delete " + name + "?"))
    return;
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    backend.deleteThing(thingID);
    emit backend->reloadEvent(); // PONG
  });
}

MyWidget::MyWidget() : backend(getBackend()) {
  connect(backend, &BackendObj::gotNameForThing, this, &MyWidget::askDelete);

}
```


revng

```cpp
void MyWidget::deleteThing(int thingID) {
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty())
      return; // Thing not found
    emit backend->gotNameForThing(result[0]->getName()); // PONG
  });
}

void MyWidget::askDelete(QString name) {
  if(!askQuestion("Do you want to delete " + name + "?"))
    return;
  ThreadPool::pushTask([thingID] (Backend &backend) { // PING
    backend.deleteThing(thingID);
    emit backend->reloadEvent(); // PONG
  });
}

MyWidget::MyWidget() : backend(getBackend()) {
  connect(backend, &BackendObj::gotNameForThing, this, &MyWidget::askDelete);
  connect(backend, &BackendObj::reloadEvent, this, &MyWidget::reloadViews);
}
```


rev.ng

Pros	Cons
• Lifetime checks are not delegated to programmer because Qt disconnects signals when the object's lifetime ends • Extensible (even too much), just connect more signals and slots	• Asynchronous but sequential logic of operation is scattered around the code • Qt-specific

```cpp
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING









  });
}
```

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found















  });
}
```


revng

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG











    });
  });
}
```


rev.ng

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG
      if (p.isNull()) return; // Check if window was closed











    });
  });
}
```


revng

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG
      if (p.isNull()) return; // Check if window was closed

      if(!p->askQuestion("Do you want to delete " + name + "?"))
        return; // User answers 'No'







    });
  });
}
```


revng

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG
      if (p.isNull()) return; // Check if window was closed

      if(!p->askQuestion("Do you want to delete " + name + "?"))
        return; // User answers 'No'

      // User answers 'Yes'
      ThreadPool::pushTask([thingID, p](Backend &backend) { // PING


      });
    });
  });
}
```

```cpp
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG
      if (p.isNull()) return; // Check if window was closed

      if(!p->askQuestion("Do you want to delete " + name + "?"))
        return; // User answers 'No'

      // User answers 'Yes'
      ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
        backend.deleteThing(thingID);

      });
    });
  });
}
```


rev.ng

```
void MyWidget::deleteThing(int thingID) {
  auto p = QPointer<MyWidget>(this);
  ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
    auto result = backend.findThing(thingID);
    if (result.empty()) return; // Thing not found

    std::string name = result[0]->getName();
    dispatchToMainThread([name, thingID, p]() { // PONG
      if (p.isNull()) return; // Check if window was closed

      if(!p->askQuestion("Do you want to delete " + name + "?"))
        return; // User answers 'No'

      // User answers 'Yes'
      ThreadPool::pushTask([thingID, p](Backend &backend) { // PING
        backend.deleteThing(thingID);
        dispatchToMainThread([p]() { if (!p.isNull()) p->reloadViews(); }); // PONG
      });
    });
  });
}
```

Pros	Cons

• Asynchronous sequential flow is clearly visible in the code
• Plain portable C++, not Qt or other framework
• Extensible (with some boilerplate)
	• Checks delegated to programmer
	  ◦ prevent use in worker thread of non-thread-safe objects owned by GUI/main thread
	  ◦ when back to GUI, check lifetime of objects to notify results
	• Poor clarity/readability
	  ◦ Nested lambdas force verbose captures and argument passing
	  ◦ Indentation rapidly becomes unmanageable

rev.ng

- The JavaScript promise/future way (i.e. the `.then()` operator)
- The C++ `std::promise`/`std::future` way
- Other big projects/framework roll their own (e.g. Chromium)

They all suffer of some combination of the previous drawbacks

revng

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

- Coroutines are a generalization of the concept of subroutines
- According to Donald Knuth, invented by Melvin E. Conway in 1958 [1]
- TLDR: like subroutines, but their execution can be suspended and resumed

[1] Conway, Melvin E. (July 1963). "Design of a Separable Transition-diagram Compiler"

revng

– Subroutines –

– Subroutines –

- Call

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning

rev.ng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning
  - subsequently from suspension point

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning
  - subsequently from suspension point
- Suspend execution (saving information in coroutine state)

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning
  - subsequently from suspension point
- Suspend execution (saving information in coroutine state)
  - returning a value to caller (`yield`)

revng

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning
  - subsequently from suspension point
- Suspend execution (saving information in coroutine state)
  - returning a value to caller (`yield`)
  - not returning a value to caller (`await`)

– Subroutines –

- Call
  - Creates subroutine context (stack frame)
  - Starts execution (resume from beginning)
  - Executes until `return`
- Return
  - Returns control back to caller
  - Returns value back to caller (optional)
  - Destroys subroutine context (stack frame)

– Coroutines –

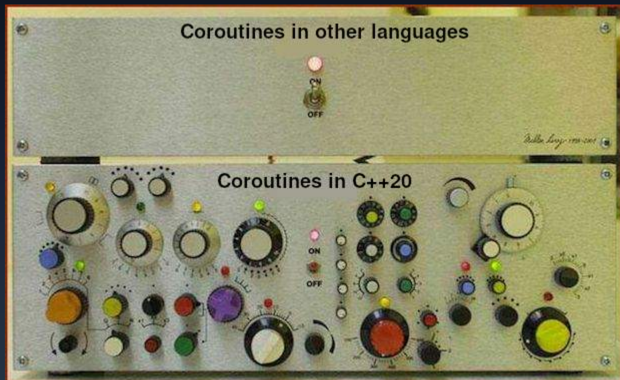
- Create coroutine state (on the heap in C++)
- Resume execution
  - initially from beginning
  - subsequently from suspension point
- Suspend execution (saving information in coroutine state)
  - returning a value to caller (`yield`)
  - not returning a value to caller (`await`)
- Destroy coroutine state

rev.ng

- Suspension returning value: `yield` (not seen in detail in this talk)
- Suspension without returning value: `await`

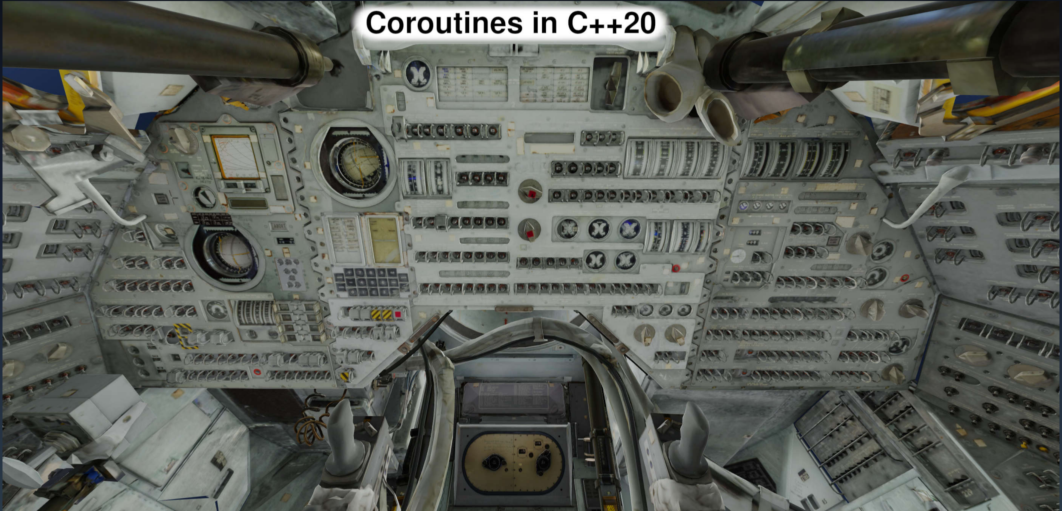
```
def MyCoroutine():
    for i in [1..10]:
        print(i);
        await;

def Main():
    printer = MyCoroutine::create();

    while not printer.done():
        printer.resume();
```

Output: 1 2 3 4 5 6 7 8 9.



revng

revng



rev.ng

- Available from C++20
- Support is still experimental in major compilers (gcc, clang, ...)
- Thins library support yet: you have to roll your own classes
- The syntax is similar to a plain old functions
- The are a bunch of types and objects associated with a coroutine
- The standard imposes requirements for a coroutine and its associated types

revng

Each coroutine is associated with the following notable objects.

Each coroutine is associated with the following notable objects.

Coroutine State. Contains the coroutine execution context (e.g. arguments, local variables, program counter). Never accessed directly, only by the language under the hood.

rev.ng

Each coroutine is associated with the following notable objects.

Coroutine State. Contains the coroutine execution context (e.g. arguments, local variables, program counter). Never accessed directly, only by the language under the hood.

Promise Object. Used from within the coroutine (e.g. to suspend and yield results to the caller). It is user-defined, but the languages requires a given interface.

rev.ng

Each coroutine is associated with the following notable objects.

Coroutine State. Contains the coroutine execution context (e.g. arguments, local variables, program counter). Never accessed directly, only by the language under the hood.

Promise Object. Used from within the coroutine (e.g. to suspend and yield results to the caller). It is user-defined, but the languages requires a given interface.

Coroutine Handle. Non-owning handle used from outside the coroutine (e.g. to control the execution of the coroutine, resume it, destroy it). One of the few types currently provided by the standard library: `std::coroutine_handle<T>`.

revng

- `co_await`: suspension point not returning a value
- `co_yield`: suspension point with return value
- `co_return`: end of execution

If a function uses any of these, then it is a candidate coroutine

To emit coroutine code, the compiler needs a bunch of other things

```cpp
using awaitable = std::suspend_always;
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;
      co_await awaitable();
  }
}
```

Associated types and objects

- Resumable
- Coroutine Handle
- Awaitable Types

revng

```
using awaitable = std::suspend_always;
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;
      co_await awaitable();
  }
}
```

Associated types and objects

- Resumable
  - must be a class
  - must expose a public type called promise_type
  - promise_type is the type the Promise Object for MyCoroutine
  - promise_type needs to expose a given interface (not important for this talk)
- Coroutine Handle
- Awaitable Types

revng

```cpp
using awaitable = std::suspend_always;
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;
      co_await awaitable();
  }
}
```

Associated types and objects

- `Resumable`
- Coroutine Handle
  - Its type is retrieved by the language
  - STL provides `std::coroutine_handle<Resumable::promise_type>`
  - Exposes methods such as: `promise()`, `done()`, `destroy()`
- Awaitable Types

revng

```cpp
using awaitable = std::suspend_always;
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;
      co_await awaitable();
  }
}
```

Associated types and objects

- `Resumable`
- Coroutine Handle
- Awaitable Types
    - `co_await` is called against them
    - very important for this talk

revng

```
class Awaiter {
    using coro_handle = std::coroutine_handle<Resumable::promise_type>;

 public:
    // Return values: false, suspend; true, continue.
    bool await_ready();
```


revng

```cpp
class Awaiter {
    using coro_handle = std::coroutine_handle<Resumable::promise_type>;

 public:
    // Return values: false, suspend; true, continue.
    bool await_ready();

    // Return values: true, suspends coroutine; false, continues execution
    bool await_suspend(coro_handle CH);
```


revng

```cpp
class Awaiter {
    using coro_handle = std::coroutine_handle<Resumable::promise_type>;

 public:
    // Return values: false, suspend; true, continue.
    bool await_ready();

    // Return values: true, suspends coroutine; false, continues execution
    bool await_suspend(coro_handle CH);

    // Result of the awaitable object
    AwaitResult await_resume();
};
```

```cpp
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;

      co_await std::suspend_always();








  }
}
```

```cpp
Resumable MyCoroutine() {
  for (int i = 1; i < 10; i++) {
      std::cout << i;

      auto TheAwaitable = std::suspend_always();
      if (not TheAwaitable.await_ready()) {
        if (await_suspend(handle)) {
          // execution is suspended here
          // execution restarts here on next resume
        }
      }

      [[maybe_unused]]
      auto var = TheAwaitable.await_resume();
  }
}
```


revng

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

- Encourage using local sequential syntax
- Minimize boilerplate for asynchronous code and thread migration
- Use just standard C++, to make it seamlessly portable
- Avoid delegating checks to the programmer

1. Self-Dispatching Coroutines

rev.ng

1. Self-Dispatching Coroutines
2. Automate lifetime checks on resume

1. Self-Dispatching Coroutines
2. Automate lifetime checks on resume
3. Automatically swap thread-unsafe frontend pointers with `nullptr` on suspend

revng

1. Self-Dispatching Coroutines
2. Automate lifetime checks on resume
3. Automatically swap thread-unsafe frontend pointers with `nullptr` on suspend
4. Automatically guard backend from access from frontend

rev.ng

Coroutines that switch from main thread to thread pool and back on `co_await`

revng

Coroutines that switch from main thread to thread pool and back on `co_await`

```
Resumable myCoroutine() {
    // Execution is started from the main thread




}
```


revng

Coroutines that switch from main thread to thread pool and back on co_await

```
Resumable myCoroutine() {
    // Execution is started from the main thread
    co_await ToThreadPool(); // <- Switches thread under the hood



}
```


rev.ng

Coroutines that switch from main thread to thread pool and back on co_await

```
Resumable myCoroutine() {
    // Execution is started from the main thread
    co_await ToThreadPool(); // <- Switches thread under the hood
    // Executions continues on the thread pool


}
```


rev.ng

Coroutines that switch from main thread to thread pool and back on `co_await`

```
Resumable myCoroutine() {
    // Execution is started from the main thread
    co_await ToThreadPool(); // <- Switches thread under the hood
    // Executions continues on the thread pool
    co_await ToMainThread(); // <- Switches thread under the hood

}
```

Coroutines that switch from main thread to thread pool and back on `co_await`

```
Resumable myCoroutine() {
    // Execution is started from the main thread
    co_await ToThreadPool(); // <- Switches thread under the hood
    // Executions continues on the thread pool
    co_await ToMainThread(); // <- Switches thread under the hood
    // Execution is back to main thread
}
```


revng

Coroutines that switch from main thread to thread pool and back on `co_await`

```
Resumable myCoroutine() {
    // Execution is started from the main thread
    co_await ToThreadPool(); // <- Switches thread under the hood
    // Executions continues on the thread pool
    co_await ToMainThread(); // <- Switches thread under the hood
    // Execution is back to main thread
}
```

We need to bake the thread-switch logic into custom awaitables: `ToThreadPool` and `ToMainThread`

rev.ng

```
struct ToThreadPool {
    bool await_suspend(coro_handle CH) {
        ThreadPool::pushTask([CH]() {
            Resumable(CH).resume();
        });
        return true; // Suspend!!
    }
};

struct ToMainThread {
    bool await_suspend(coro_handle CH) {
        dispatchToMainThread([CH]() {
            Resumable(CH).resume();
        });
        return true; // Suspend!!
    }
};
```

- co_await calls await_suspend, which confirms the suspend
- await_suspend reschedules the coroutine in a new thread
- the execution is resumed in the new thread

revng

```
Resumable MyWidget::deleteThing(int thing_id) {
    QPointer<MyWidget> p(this);
    co_await ToThreadPool(); // *** Switch to Thread Pool ***









}
```


rev.ng

```
Resumable MyWidget::deleteThing(int thing_id) {
    QPointer<MyWidget> p(this);
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    auto result = getBackend()->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    co_await ToMainThread(); // *** Switch to Main Thread ***









}
```


revng

```cpp
Resumable MyWidget::deleteThing(int thing_id) {
    QPointer<MyWidget> p(this);
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    auto result = getBackend()->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    co_await ToMainThread(); // *** Switch to Main Thread ***
    if (p.isNull()) co_return;
    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    // User answers yes.
    co_await ToThreadPool(); // *** Switch to Thread Pool ***




}
```


revng

```cpp
Resumable MyWidget::deleteThing(int thing_id) {
    QPointer<MyWidget> p(this);
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    auto result = getBackend()->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    co_await ToMainThread(); // *** Switch to Main Thread ***
    if (p.isNull()) co_return;
    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    // User answers yes.
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    getBackend()->deleteThing(thing_id);

    co_await ToMainThread(); // *** Switch to Main Thread ***


}
```


rev.ng

```
Resumable MyWidget::deleteThing(int thing_id) {
    QPointer<MyWidget> p(this);
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    auto result = getBackend()->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    co_await ToMainThread(); // *** Switch to Main Thread ***
    if (p.isNull()) co_return;
    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    // User answers yes.
    co_await ToThreadPool(); // *** Switch to Thread Pool ***
    getBackend()->deleteThing(thing_id);

    co_await ToMainThread(); // *** Switch to Main Thread ***
    if (!p.isNull())
        p->reloadViews();
}
```


revng

Pros	Cons

• Local sequential syntax, with visible flow
• No boilerplate
  • thread switching on `co_await`, thanks to custom awaitables (`ToThreadPool` and `ToMainThread`)
  • stack variables are shared between threads
• Easily extensible
• Uses standard C++ syntax
	• Checks are delegated to the programmer

revng

Programmer still has to check for lifetime of GUI objects

```
Resumable MyWindow::myMethod() {
    QPointer<QPushButton> btn = this->findChild<QPushButton*>();

    co_await ToThreadPool(); // Switch to worker thread
    doSomething(); // User closes window while worker is doing this

    co_await ToMainThread(); // Switch to main thread

    // Missing: if(btn.isNull()) return;

    btn->setText("Crash Here!");
    co_return;
}
```

How can we force and automate lifetime checks?

```
Resumable MyWindow::myMethod() {
    QPushButton* btn = this->findChild<QPushButton*>();
    Guard guard(btn); // We create a guard object to wrap variables to check

    co_await ToThreadPool(); // Switch to worker thread
    doSomething();  // User closes window while worker is doing this

    if (!co_await guard.toMainThread()) // Switch to main thread
        co_return; // Exit if the window is closed

    btn->setText("Doesn't crash here :)"); // Checks are automated in co_await
    co_return;
}
```


revng

```
template<typename... Args> class Guard {
    std::tuple<safe_reference_t<Args>...> saferef; // Refs to variables to check

public:
    Guard(Args &... t) : saferef(t...) { }









};
```

```
template<typename... Args> class Guard {
    std::tuple<safe_reference_t<Args>...> saferef; // Refs to variables to check

public:
    Guard(Args &... t) : saferef(t...) { }

    // Check method
    bool checkAlive() const {
      // Checks if all variable refs in CheckableTuple are alive.
      // Uses our safe_reference_t<T> trait, who exposes methods for checks.
      // We specialized safe_reference_t<T> for std::shared_ptr and QPointer.
    }



};
```


rev.ng

```cpp
template<typename... Args> class Guard {
    std::tuple<safe_reference_t<Args>...> saferef; // Refs to variables to check

public:
    Guard(Args &... t) : saferef(t...) { }

    // Check method
    bool checkAlive() const {
      // Checks if all variable refs in CheckableTuple are alive.
      // Uses our safe_reference_t<T> trait, who exposes methods for checks.
      // We specialized safe_reference_t<T> for std::shared_ptr and QPointer.
    }

    // Factory methods for awaitables
    ToMainThread toMainThread() { return ToMainThread(*this); }
    ToThreadPool toThreadPool() { return ToThreadPool(*this); }
};
```


revng

The awaitable is created with the check in the `await_resume()`: `await_resume` of `ToMainThread` returns a `[[nodiscard]] bool`, which forces checks at compile time.

```
template<typename... Args>
class ToMainThread
{
    Guard<Args...> &guard;
public:
    ToMainThread(Guard<Args...> &guard) : guard(guard) {}

    bool await_suspend(coro_handle CH) {
        // Same as before
        dispatchToMainThread([CH]() { Resumable(CH).resume(); });
        return true; // Suspend!! Will be resumed in main thread
    }




};
```


rev.ng

The awaitable is created with the check in the `await_resume()`: `await_resume` of `ToMainThread` returns a `[[nodiscard]] bool`, which forces checks at compile time.

```
template<typename... Args>
class ToMainThread
{
    Guard<Args...> &guard;
public:
    ToMainThread(Guard<Args...> &guard) : guard(guard) {}

    bool await_suspend(coro_handle CH) {
        // Same as before
        dispatchToMainThread([CH]() { Resumable(CH).resume(); });
        return true; // Suspend!! Will be resumed in main thread
    }

    // await_resume checks the guard when coroutine is resumed!
    // Its result cannot be discarded. Compile-time warning forces user to check!
    [[nodiscard]] bool await_resume() { return guard.checkAlive(); }
};
```


revng

```
Resumable MyWidget::deleteThing(int thing_id) {
  MyWidget *p = this;
  Guard guard(p);
  co_await guard.toThreadPool(); // Switch to Thread Pool
  auto result = getBackend()->findThing(thing_id);
  if (result.empty()) co_return;
  std::string curName = result[0]->getName();

  // Switch to Main Thread, with automated checks, and forcing user to use the result
  if (!co_await guard.toMainThread())
      co_return;
  if (!p->askQuestion("Do you want to delete " + curName + "?"))
      co_return; // User answers no.

  // User answers yes.
  co_await guard.toThreadPool(); // Switch to Thread Pool
  getBackend()->deleteThing(thing_id);

  // Switch to Main Thread, with automated checks, and forcing user to use the result
  if (co_await guard.toMainThread())
      p->reloadViews();
}
```


revng

Pros	Cons

• **NEW**: Lifetime checks are automated on `co_await`, and user is forced at compile-time to check result
• Local sequential syntax, with visible flow
• No boilerplate
  ◦ thread switching on `co_await`, thanks to custom awaitables (`ToThreadPool` and `ToMainThread`)
  ◦ stack variables are shared between threads
• Easily extensible
• Uses standard C++ syntax
	• Some checks are still delegated to the programmer: thread-unsafe variables can be accessed in the wrong thread

revng

Accessing a GUI variable from worker threads causes undefined behavior.

```
Resumable MyWindow::myMethod() {
    QPushButton* btn = this->findChild<QPushButton*>();
    Guard guard(btn);

    co_await guard.toThreadPool(); // Switch to Thread Pool

    btn->setText("May crash here"); // Race-condition! QPushButton is not thread-safe!

    co_return;
}
```

Can we do something about that?

Yes! Swap all unsafe pointers with `nullptr` before performing a thread switch

```cpp
class Guard {
    // ... as before

    Guard(Args &... t) : saferef(t...), ptrs(t...) { }
private:
    std::tuple<Args &...> ptrs;
    std::tuple<Args...> memory; // Initialized with null pointers

    // Swaps pointers referenced by 'ptrs' with 'memory'
    void swapContext();
}
```


revng

Factory methods for awaitables are extended to swap the pointers

```
class Guard {
  // ...
public:
  ToThreadPool toThreadPool() {
    swapContext<TupleSize>();
    // Here pointers to frontend objects are nullified
    return ToThreadPool(*this);
  }

  ToMainThread toMainThread() {
    swapContext<TupleSize>();
    // Here pointers to frontend objects are restored
    return ToMainThread(*this);
  }
};
```


rev.ng

```cpp
Resumable MyWidget::deleteThing(int thing_id) {
  MyWidget *p = this;
  Guard guard(p);
  co_await guard.toThreadPool(); // Now swaps p with nullptr
  auto result = getBackend()->findThing(thing_id); // Using p here always crashes
  if (result.empty()) co_return;
  std::string curName = result[0]->getName();

  // Switch to Main Thread, with automated checks, and forcing user to use the result
  if (!co_await guard.toMainThread()) // <- This now swaps p back in place
      co_return;
  if (!p->askQuestion("Do you want to delete " + curName + "?")) // <- p is safe to use
      co_return;

  co_await guard.toThreadPool(); // Now swaps p with nullptr
  getBackend()->deleteThing(thing_id); // Using p here always crashes

  // Switch to Main Thread, with automated checks, and forcing user to use the result
  if (co_await ToMainThread()) // <- This now swaps p back in place
    p->reloadViews(); // <- p is safe to use
}
```


revng

Pros	Cons

• **NEW**: Thread-Unsafe frontend pointers are automatically checked
• Lifetime checks are automated on `co_await`
• Local sequential syntax, with visible flow
• No boilerplate
  ◦ thread switching on `co_await`, thanks to custom awaitables (`ToThreadPool` and `ToMainThread`)
  ◦ stack variables are shared between threads
• Easily extensible
• Uses standard C++ syntax
	• Some checks are still delegated to the programmer: thread-unsafe backend pointers are still unchecked

rev.ng

- We need to secure `getBackend()` from uses in the main thread
- `getBackend()` is only safe to access on the thread pool
- Again, use awaitables to automate logic on suspend/resume
- Remember, `co_await` can return a value to the coroutine after resume

revng

```
Resumable MyWidget::deleteThing(int thing_id) {
    MyWidget* p(this);
    Guard guard(p);
    // *** Switch to Thread Pool ***
    co_await guard.toThreadPool();
    auto result = getBackend()->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    if (!co_await guard.toMainThread()) // *** Switch to Main Thread ***
        co_return;

    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    co_await guard.toThreadPool(); // *** Switch to Thread Pool ***
    getBackend()->deleteThing(thing_id);

    if (co_await guard.toMainThread()) // *** Switch to Main Thread ***
        p->reloadViews();
}
```


rev.ng

```
Resumable MyWidget::deleteThing(int thing_id) {
    MyWidget* p(this);
    Guard guard(p);
    // *** Switch to Thread Pool ***
    ExpiringPtr<Backend> backend = co_await guard.toThreadPool();
    auto result = backend->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    if (!co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
        co_return;

    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    backend = co_await guard.toThreadPool(); // *** Switch to Thread Pool ***
    backend->deleteThing(thing_id);

    if (co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
        p->reloadViews();
}
```


rev.ng

```
template <typename T>
class ExpiringPtr {
public:
    ExpiringPtr(T *F) : F(F) , Expired(false) {}
    T *operator->() const { assert(!Expired); return F; }
    T &operator*() const { assert(!Expired); return *F; }
    operator bool() const { assert(!Expired); return F != nullptr; }
    void markExpired() { assert(!Expired); Expired = true; }

private:
    T *F;
    bool Expired;
};
```


rev.ng

Remember that: the `co_await` can also return some value or object.
This is the return value of the `await_resume`.

```
class ToThreadPool {
public:
    bool await_suspend(coro_handle CH) {
        // Like before
        ThreadPool::pushTask([CH]() {
            Resumable(CH).resume();
        });
        return true; // Suspend!! Will be resumed on main
    }

    ExpiringPtr<Backend> await_resume() {
        // When resuming, returns a new ExpiringPtr to the coroutine
        return ExpiringPtr<Backend>(getBackend());
    }
};
```


rev.ng

The awaitable ToMainThread wants a valid ExpiringPtr<Backend> to be constructed, and expires it.

```
class ToMainThread
{
    Guard<Args...> &guard;
public:
    // Constructing a ToMainThread now requires a backend pointer,
    // which is invalidated, so that accessing it from main crashes
    ToMainThread(Guard<Args...> &g,
                 ExpiringPtr<Backend> &expiringBackend) : guard(g) {
     expiringBackend.markExpired();
   }

    bool await_suspend(coro_handle CH) { /* same as before */ }
};
```


revng

```
Resumable MyWidget::deleteThing(int thing_id) {
    MyWidget* p(this);
    Guard guard(p);
    // *** Switch to Thread Pool ***
    ExpiringPtr<Backend> backend = co_await guard.toThreadPool();
    auto result = backend->findThing(thing_id);
    if (result.empty()) co_return;
    std::string curName = result[0]->getName();

    if (!co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
        co_return;

    if (!p->askQuestion("Do you want to delete " + curName + "?"))
        co_return; // User answers no.

    backend = co_await guard.toThreadPool(); // *** Switch to Thread Pool ***
    backend->deleteThing(thing_id);

    if (co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
        p->reloadViews();
}
```


rev.ng

Introduction and Motivation

Some Examples of Approaches

Coroutines

Proposed Solution: Self-Dispatching Coroutines

Conclusion

rev.ng

- Pure standard C++20, not limited to Qt
- Sequential syntax with asynchronous execution – [thanks coroutines!!]
- Stack variables are shared between threads
  - great readability
  - no boilerplate to pass data around
- Checks are not delegated to programmer – [thanks awaitables!!]
  - awaitables wrap lifetime checks in `await_resume`
  - awaitables swap non-thread-safe ptrs with `nullptr` on suspension points
- Very easy to extend with more ping-pong steps
  - add code on the bottom
  - call `co_await` where needed to switch threads
  - you can even use control-flow constructs (ifs, loops)

```
Resumable myCoroutine(...) {
    Guard guard(...);
    // *** Switch to Thread Pool ***
    ExpiringPtr<Backend> backend = co_await guard.toThreadPool();

    if (backend->isOk()) {
      // Do something on thread pool ...
      if (!co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
          co_return;
      // Do something on main thread ...
      // *** Switch back to Thread Pool ***
      backend = co_await guard.toThreadPool();
    } else {
      // Do something else on thread pool ...
      if (!co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
          co_return;
      // Do something else main thread ...
      // *** Switch back to Thread Pool ***
      backend = co_await guard.toThreadPool();
    }
}
```


rev.ng

```
Resumable myCoroutine(...) {
    Guard guard(...);
    // *** Switch to Thread Pool ***
    ExpiringPtr<Backend> backend = co_await guard.toThreadPool();

    while (backend->isCool()) {
      // Do something on thread pool ...
      if (!co_await guard.toMainThread(backend)) // *** Switch to Main Thread ***
          co_return;
      // Do something on main thread ...
      // *** Switch back to Thread Pool ***
      backend = co_await guard.toThreadPool();
    }
}
```


rev.ng

- Heap traffic for allocating coroutine states
  - can be optimized away from the compiler
- Swaps to avoid race conditions are executed every time
  - can be turned off in production
- Currently the compiler support is still poor
- The debugging support is even worse

Thanks!

Get in touch at:

`info@rev.ng`

We are hiring!

Subscribe to our newsletter:

`https://rev.ng/newsletter.html`

Questions?