Coroutines As An API Principle

Meeting C++, November 11th, 2021

Marc Mutz <marc.mutz@qt.io>

- **Introduction**
- Recap: Non-Owning Interface Idiom (NOIv1)
- Non-Owning Interface Idiom v2 (NOIv2)
- Summary: Coroutines As An API Principle

Why this talk?

This is not a talk about *how* to write coroutines...

- ...but *why*

There are plenty of resources out there that deal with the *how*...

- ...just enter "Gor Nishanov coroutines" into YouTube :)

Why this talk? (cont'd)

I just deeply dislike owning containers in APIs:

- Qt is full of APIs that take and return owning containers:
  - `QLabel::setText(QString)`
    - what if my backend gives me `std::u16string s16`?
      - `setText(QString::fromStdU16String(s16))`
    - Or `pmr::u16string ps16`?
      - `setText(QString::fromUtf16(ps16.data(), ps16.size()))` (inkl. signed/unsigned mismatch)
  - `QVector<QGradientStop> QGradient::stops()`
    - what if I just need the first two stops?
  - `QVector<QRect> QRegion::rects()`
    - What if `QRegion` wants to use SBO internally? (hint: it does)
- But the standard library is also not free of this:
  - `std::string std::error_category::message(int)`
    - But what if I want to use `error_category` in Freestanding implementations?
      - Oops, no `std::string` in Freestanding...

Problem 1: Owning Containers in APIs hinder interoperability:

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const;
4      ~~~~
5  };
6
7  void fillRects(const std::vector<QRect>& rects);
8
9  QRegion r = ~~~~;
10 fillRects(r.rects()); // OOPS
```

Problem 1: Owning Containers in APIs hinder interoperability:

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const;
4      ~~~~
5  };
6
7  void fillRects(const std::vector<QRect>& rects);
8
9  QRegion r = ~~~~;
10 fillRects(r.rects()); // OOPS
```

Solution: Non-Owning Interface Idiom (NOIv1): Use views for function arguments:

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const;
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8
9  QRegion r = ~~~~;
10 fillRects(r.rects()); // OK (except that std::span is broken, and this doesn't compile)
```

Problem 2: Owning Containers in APIs restrict the implementation

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const; // may allocate
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8
9  QRegion r = ~~~~;
10 fillRects(r.rects()); // OK, but may throw
```

Problem 2: Owning Containers in APIs restrict the implementation

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const; // may allocate
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8
9  QRegion r = ~~~~;
10 if (r.numRects() == 1) {
11     auto rect = r.boundingRect(); // OK, doesn't allocate
12     fillRects({&rect, 1});
13 } else {
14     fillRects(r.rects()); // OK, doesn't allocate
15 }
```

Problem 2: Owning Containers in APIs restrict the implementation

```
1  class QRegion {
2      ~~~~
3      QVector<QRect> rects() const; // may allocate
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8
9  QRegion r = ~~~~;
10 if (r.numRects() == 1) {
11     auto rect = r.boundingRect(); // OK, doesn't allocate
12     fillRects({&rect, 1});
13 } else {
14     fillRects(r.rects()); // OK, doesn't allocate
15 }
```

Solution: Non-Owning Interface Idiom (NOIv1): Use views for function return values:

```
1  class QRegion {
2      ~~~~
3      std::span<QRect> rects() const noexcept; // never allocates
4      ~~~~
5  };
6
7  QRegion r = ~~~~;
8  fillRects(r.rects()); // OK, simple, yet never allocates
```

Issue 1: NOI creates lifetime issues

```
1  class QRegion {
2      ~~~~
3      std::span<QRect> rects() const noexcept;
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8  QRegion getRegion();
9
10 fillRects(getRegion().rects()); // OK
11 const auto rects = getRegion().rects();
12 fillRects(rects); // OOPS, dangling reference
```

Issue 1: NOI creates lifetime issues

```
1  class QRegion {
2      ~~~~
3      std::span<QRect> rects() const noexcept;
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8  QRegion getRegion();
9
10 fillRects(getRegion().rects()); // OK
11 const auto rects = getRegion().rects();
12 fillRects(rects); // OOPS, dangling reference
```

(Wrong) Solution: delete the rvalue overload

```
1  class QRegion {
2      ~~~~
3      std::span<QRect> rects() const & noexcept;
4      void rects() const && = delete;
5      ~~~~
6  };
7
8  void fillRects(std::span<QRect> rects);
9  QRegion getRegion();
10
11 fillRects(getRegion().rects()); // ERROR (but was OK!)
12 const auto rects = getRegion().rects(); // ERROR (good)
```

Next Try: Issue 1: NOI creates lifetime issues

```
1 fillRects(getRegion().rects()); // OK
2 const auto rects = getRegion().rects();
3 fillRects(rects); // OOPS, dangling reference
```

Next Try: Issue 1: NOI creates lifetime issues

```
1  fillRects(getRegion().rects()); // OK
2  const auto rects = getRegion().rects();
3  fillRects(rects); // OOPS, dangling reference
```

Solution: implement a Clang/Clazy checker

```
1  class QRegion {
2      ~~~~
3      std::span<QRect> rects() const noexcept;
4      ~~~~
5  };
6
7  void fillRects(std::span<QRect> rects);
8  QRegion getRegion();
9
10 fillRects(getRegion().rects()); // OK
11 const auto rects = getRegion().rects(); // ERROR: clazy-dangling-view
```

See also Herb Sutter's `-Wlifetime` work:

- https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Issue 2: views only work with contiguous containers

```
1 class QAbstractItemModel {
2     ~~~~
3     virtual QHash<Q::ItemRole, QByteArray> roleNames() const; // non-contiguous
4     ~~~~
5 };
```

Issue 2: views only work with contiguous containers

```
1 class QAbstractItemModel {
2     ~~~~
3     virtual QHash<Q::ItemRole, QByteArray> roleNames() const; // non-contiguous
4     ~~~~
5 };
```

Issue 3: views only work when there's backing storage

```
1 class QString {
2     ~~~~
3     QVector<QStringView> split(QStringView sep) const; // contiguous, but calculated
4     ~~~~
5 };
```

Issue 2: views only work with contiguous containers

```
1 class QAbstractItemModel {
2     ~~~~
3     virtual QHash<Q::ItemRole, QByteArray> roleNames() const; // non-contiguous
4     ~~~~
5 };
```

Issue 3: views only work when there's backing storage

```
1 class QString {
2     ~~~~
3     QVector<QStringView> split(QStringView sep) const; // contiguous, but calculated
4     ~~~~
5 };
```

Solution: NOIv2: use coroutines

- Introduction
- Recap: Non-Owning Interface Idiom (NOIv1)
- **Non-Owning Interface Idiom v2 (NOIv2)**
- Summary: Coroutines As An API Principle

Problem 3: Owning Containers in APIs cause eager calculations

```
1  class QString {
2      ~~~~
3      QVector<QStringView> split(QStringView sep) const; // will split whole string
4      ~~~~
5  };
6
7  QString s = ~~~~;
8  for (auto part : s.split(u"\r\n")) {
9      use(part);
10     if (part.trimmed() == u"EOF") // what if a 100M file contains this as the 3rd line?
11         break;
12 }
```

Problem 3: Owning Containers in APIs cause eager calculations

```
1  class QString {
2      ~~~~
3      QVector<QStringView> split(QStringView sep) const; // will split whole string
4      ~~~~
5  };
6
7  QString s = ~~~~;
8  for (auto part : s.split(u"\r\n")) {
9      use(part);
10     if (part.trimmed() == u"EOF") // what if a 100M file contains this as the 3rd line?
11         break;
12 }
```

Solution: Non-Owning Interface Idiom v2: Use a coroutine:

```
1  class QString {
2      ~~~~
3      std::generator<QStringView> split(QStringView sep) const; // lazy sequence
4      ~~~~
5  };
6
7  QString s = ~~~~;
8  for (auto part : s.split(u"\r\n")) {
9      use(part);
10     if (part.trimmed() == u"EOF") // simple, but efficient (lazy)
11         break;
12 }
```

Case Study: `QString::split()` (cont'd)

Adjusting the implementation is trivial:

Case Study: `QString::split()` (cont'd)

Adjusting the implementation is trivial:

```
 1 QVector       <QStringView> QString::split(QStringView sep) const {
 2     QVector<QStringView> result;
 3     qsizetype start = 0;
 4     qsizetype end;
 5     while ((end = indexOf(sep, start)) != npos) {
 6         result.push_back(QStringView{*this}.substr(start, end - start));
 7         start = end + sep.size();
 8     }
 9     result.push_back(QStringView{*this}.substr(start));
10     return result;
11 }
```

Case Study: `QString::split()` (cont'd)

Adjusting the implementation is trivial:

```
 1 std::generator<QStringView> QString::split(QStringView sep) const {
 2
 3     qsizetype start = 0;
 4     qsizetype end;
 5     while ((end = indexOf(sep, start)) != npos) {
 6         co_yield         QStringView{*this}.substr(start, end - start) ;
 7         start = end + sep.size();
 8     }
 9     co_yield         QStringView{*this}.substr(start) ;
10
11 }
```

Case Study: `QString::split()` (cont'd)

Unlike `QStringTokenizer`, which is ~450 LOC around this central function:

Case Study: `QString::split()` (cont'd)

Unlike `QStringTokenizer`, which is ~450 LOC around this central function:

```
1  auto QStringTokenizerBase<Haystack, Needle>::next(tokenizer_state state) const noexcept -> next_result
2  {
3      if (state.end < 0) {
4          // already at end:
5          return {{}, false, state};
6      }
7      state.end = m_haystack.indexOf(m_needle, state.start);
8      Haystack result;
9      if (state.end >= 0) {
10         // token separator found => return intermediate element:
11         result = m_haystack.substr(state.start, state.end - state.start);
12         const auto ns = QtPrivate::Tok::size(m_needle);
13         state.start = state.end + ns;
14     } else {
15         // token separator not found => return final element:
16         result = m_haystack.substr(state.start);
17     }
18     return {result, true, state};
19 }
```

Case Study: `QString::split()` (cont'd)

Unlike `QStringTokenizer`, which is ~450 LOC around this central function:

```
1  auto QStringTokenizerBase<Haystack, Needle>::next(tokenizer_state state) const noexcept -> next_result
2  {
3      if (state.end < 0) {
4          // already at end:
5          return {{}, false, state};
6      }
7      state.end = m_haystack.indexOf(m_needle, state.start);
8      Haystack result;
9      if (state.end >= 0) {
10         // token separator found => return intermediate element:
11         result = m_haystack.substr(state.start, state.end - state.start);
12         const auto ns = QtPrivate::Tok::size(m_needle);
13         state.start = state.end + ns;
14     } else {
15         // token separator not found => return final element:
16         result = m_haystack.substr(state.start);
17     }
18     return {result, true, state};
19 }
```

Granted, that one has two features the coroutine `QString::split()` still lacks:

- No allocations → noexcept
- Rvalue pinning (no dangling references)

Case Study: `QString::split()` (cont'd)

Can't use Coroutines, yet?

- NP: Just throw 500 LOC at the problem :)
- or: implement a `for_each_x`:

Case Study: `QString::split()` (cont'd)

Can't use Coroutines, yet?

Case Study: `QString::split()` (cont'd)

Can't use Coroutines, yet?

- NP: Just throw 500 LOC at the problem :)

Case Study: `QString::split()` (cont'd)

Can't use Coroutines, yet?

- NP: Just throw 500 LOC at the problem :)
- or: implement a `for_each_x`:

Case Study: `QString::split()` (cont'd)

Can't use Coroutines, yet?

- NP: Just throw 500 LOC at the problem :)
- or: implement a `for_each_x`:

```
1  class QString {
2      ~~~~
3      void for_each_split_part(QStringView sep, std::function<void(QStringView)> cb) const {
4          qsizetype start = 0;
5          qsizetype end;
6          while ((end = indexOf(sep, start)) != npos) {
7              cb(QStringView{*this}.substr(start, end - start));
8              start = end + sep.size();
9          }
10         cb(QStringView{*this}.substr(start));
11     }
12     ~~~~
13 };
14
15 s.for_each_split_part([](auto part) { use(part); });
```

Issue 4: Coroutines have their own set of reference dangling problems

```cpp
std::generator<std::string_view> split(const std::string &s, char sep) {
    size_t start = 0;
    size_t end;
    while ((end = s.find(sep, start)) != std::string::npos) {
        co_yield std::string_view{s}.substr(start, end - start);
        start = end + 1;
    }
    co_yield std::string_view{s}.substr(start);
}

for (auto part : split("Hello\nWorld", '\n'))
    use(part); // oops
```

Don't we all wish the definition of ranged for loops had been like this instead?

```
for (for_range_declaration : for_range_initializer)
    statement;
```

```
[&](auto&& range) {
    auto begin = begin_expr;
    auto end = end_expr;
    for (; begin != end; ++begin) {
        for_range_declaration = *begin;
        statement;
    }
}(for_range_initializer)
```

Issue 4: Coroutines have their own set of reference dangling problems

```cpp
std::generator<std::string_view> split(const std::string &s, char sep) {
    size_t start = 0;
    size_t end;
    while ((end = s.find(sep, start)) != std::string::npos) {
        co_yield std::string_view{s}.substr(start, end - start);
        start = end + 1;
    }
    co_yield std::string_view{s}.substr(start);
}

auto parts = split("Hello\nWorld", '\n');
for (auto part : parts) // oops
    use(part);
```

Issue 4: Coroutines have their own set of reference dangling problems

```cpp
std::generator<std::string_view> split(const std::string &s, char sep) {
    size_t start = 0;
    size_t end;
    while ((end = s.find(sep, start)) != std::string::npos) {
        co_yield std::string_view{s}.substr(start, end - start);
        start = end + 1;
    }
    co_yield std::string_view{s}.substr(start);
}

auto parts = split("Hello\nWorld", '\n');
for (auto part : parts) // oops
    use(part);
```

And don't forget *this:

```cpp
QString getText();
for (auto part : getText().split(u'\n'))
    use(part);
```

Common generators `suspend_always` on `initial_suspend()`

- Can't write code in the function body to move values to coroutine frame.
- If you can, pass arguments by value, then it's automatic:

```
1  std::generator<std::string_view> split(std::string s, char sep) {
2      size_t start = 0;
3      size_t end;
4      while ((end = s.find(sep, start)) != std::string::npos) {
5          co_yield std::string_view{s}.substr(start, end - start);
6          start = end + 1;
7      }
8      co_yield std::string_view{s}.substr(start);
9  }
10
11 auto parts = split("Hello\nWorld", '\n');
12 for (auto part : parts) // OK (at the cost of an extra move)
13     use(part);
```

But we wanted to *avoid* using owners in the API...

Limits of NOIv2 (cont'd)

Common generators `suspend_always` on `initial_suspend()`

- Can't write code in the function body to move values to coroutine frame.
- If you can, pass arguments by value, then it's automatic.

Limits of NOIv2 (cont'd)

Common generators `suspend_always` on `initial_suspend()`

- Can't write code in the function body to move values to coroutine frame.
- If you can, pass arguments by value, then it's automatic.
- If not possible (e.g. `*this` before C++23), need a custom `promise_type` with a ctor matching the function's argument list
  - Ties `promise_type` to specific coroutine
    - Proliferation of `promise_types`
      - More Work (Implementing, Documenting, Learning)

Limits of NOIv2 (cont'd)

Common generators `suspend_always` on `initial_suspend()`

- Can't write code in the function body to move values to coroutine frame.
- If you can, pass arguments by value, then it's automatic.
- If not possible (e.g. `*this` before C++23), need a custom `promise_type` with a ctor matching the function's argument list
  - Ties `promise_type` to specific coroutine
    - Proliferation of `promise_types`
      - More Work (Implementing, Documenting, Learning)

Gold standard:

- Wrap a coroutine acting on just views in an ordinary function overload set that moves rvalues of owners to the return type or `promise_type`
  - See `QStringTokenizer` for how to do this
    - QStringView Diaries: Zero-Allocation String Splitting
  - Implies different return types for different functions in the overload set
    - Callers must use `auto` (or at least C++17 CTAD)

Limits of NOIv2 (cont'd)

Issue 5: Re-entrancy

- When a coroutine is suspended, other code runs
  - which may call back into the suspended coroutine
- Not a problem per-se
  - Each coroutine invocation gets their own frame
    - So they don't clobber each other's local variables
- But you do run an unbounded set of code when suspended
  - Like invoking callbacks
  - or virtual functions

Treat suspension points like you would invocations of callbacks or virtual functions

- Don't forget about the `initial_suspend()`!
- Keep resources in RAII objects across suspension points
  - Destroying a `coroutine_handle` unwinds the stack frame of the coroutine from the suspension point
    - Like exceptions, but can't be caught.

Limits of NOIv2 (cont'd)

Issue 6: Coroutine frame allocation

- Compiler allocates coroutine frame on the heap
  - function arguments
  - promise object
  - local variables
  - selected temporaries
- Compilers are sometimes able to elide the memory allocation
  - see e.g. CppCon 2016: Gor Nishanov C++ Coroutines: Under the covers
  - likely requires all code in between to be visible
    - no atomics, no out-of-line function calls
  - hard to do (in Qt, at least)
- Can be overridden by custom allocation function
  - on the promise object
  - can use a pool allocator
    - careful: coroutines could be resumed/destroyed on a different thread
      - not likely for generators, though

Bottom Line: Coroutines can't really be noexcept :(

(untested) maybe obtain storage using the CSS (Caller-Supplied Storage) Idiom?

```
my_generator my_coroutine(~~~args~~~, std::array<char, 42>&& storage = {}) {
    // place frame into `storage`?
    ~~~~
}
```

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access
  - better-than-linear complexity
- Provide additional API
  - complete EOP "efficient base" of operations

Issue 7: No random access

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access
  - better-than-linear complexity

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access
  - better-than-linear complexity
- Provide additional API

Limits of NOIv2 (cont'd)

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access
  - better-than-linear complexity
- Provide additional API
  - complete EOP "efficient base" of operations

Issue 7: No random access

- Coroutines can't "go back", only forward
  - Generator types are only Input Iterators
- What if you need more?
  - random or bidirectional access
  - better-than-linear complexity
- Provide additional API
  - complete EOP "efficient base" of operations

```
1 class QAbstractItemModel {
2     ~~~~
3     struct RoleAndName { Qt::ItemRole role; QByteArrayView name; };
4     virtual std::generator<RoleAndName> roleNames() const;     // O(N)
5     virtual QByteArrayView roleName(Qt::ItemRole role) const;  // O(1)
6     ~~~~
7 };
```

Summary: Coroutines As An API Principle

- Introduction
- Recap: Non-Owning Interface Idiom (NOIv1)
- Non-Owning Interface Idiom v2 (NOIv2)
- **Summary: Coroutines As An API Principle**

What Have We Learned?

- Coroutines can help avoid having to use owning containers in APIs
- Since coroutines are ordinary functions
  - they can be virtual functions
  - they can be DLL-exported
  - iow: they're usable in APIs
- The magic is in the return type
  - and these need to be exposed in the API, too
  - conflicting requirements:
    - fewer promise types are easier to learn, implement, document
      - but require type erasure → memory allocation
    - fine-tuning behaviour (allocation, argument pinning) requires per-coroutine `promise_types`
      - and inline functions

What Have We Learned? (cont'd)

- Coroutines can help with implementing lazy sequences
  - Less work wasted when only subset of results needed
  - Very easy implementation
    - compared to traditional techniques (e.g. `QStringTokenizer`)
  - Lazy sequences are everywhere
    - once you start looking for them

What Have We Learned? (cont'd)

- Coroutines come with their own set of issues:
  - Issues 1 + 4: Lifetime (dangling references)
  - Issues 2, 3 + 7: Either data is stored and contiguously so, then
    - you return a view and have random-access
    - or not, then you use a coroutine, which is, however, just an input range
  - Issue 5: Reentrancy
  - Issue 6: Coroutine frame allocation (no `noexcept`)

Q & A

Visit us in our sponsor booth!