Generator-based for loop

Generator ranges without coroutine overhead

Jonathan Müller — Barry Revzin

Proposal: Writing a generator range

```
struct generator123
{
    std::control_flow operator()(auto&& sink) const
    {
        std::control_flow flow;

        flow = sink(1); if (!flow) return flow;

        flow = sink(2); if (!flow) return flow;

        return sink(3);
    }
};
```

Proposal: Writing a generator range

```
struct generator123
{
    std::control_flow operator()(auto&& sink) const
    {
        sink(1).try?;
        sink(2).try?;
        return sink(3);
    }
};
```

Proposal: Using a generator range

User writes:

```
for (int x : generator123())
{
    std::print("{}\n", x);
    if (x == 2)
        break;
}
```

Proposal: Using a generator range

Compiler generates:

```
{
    auto __body = [&](int&& __element) -> std::control_flow {
        int x = __element;
        if (x == 2)
            return std::break_;
        std::printf("%d\n", x);
        return std::continue_;
    };

    auto __flow = generator123()(__body);
    (void)__flow;
}
```

Similar translation for `continue`, `return`, etc. in the loop body.

Motivation

Iterators

- **pro:** powerful (can be bidirectional, random access, …)
- **con:** verbose to implement
- **con:** performance suffers due to state machine (`concat_view::iterator` stores a `std::variant`)
- **con:** performance suffers due to the transform-filter problem (`r | transform(f) | filter(p)`, `f` invoked twice on filtered elements)

Motivation

Coroutines

- **pro:** trivial to implement
- **con:** input ranges only
- **con:** performance suffers due to heap allocation of coroutine state (if not elided)
- **con:** performance suffers due to (compiler-generated) state machine (resuming requires a dispatch)
- **con:** inability to yield from nested scope (e.g. `co_yield` from a lambda)

Motivation

Generator range

- **pro:** trivial to implement
- **pro:** no performance loss due to state machine
- **con:** range-based `for` loop only (cannot pause iteration and resume later)

think-cell's Experience

Context: heavy use of ranges, even for things like string formatting

```
tc::concat("The answer is "_tc, tc::as_dec(42), ".\n"_tc);
```

- use of generator-range idiom
- all relevant range adaptors implement generator range interface
- all relevant algorithms support generator ranges

But: cannot use range-based `for` loop.

Performance

`view::concat`

Methodology: 3 `std::vector<int>`, 30 elements each, sum them

- Generator range: `11 ns`
- `views::concat`: `20 ns` (2x slower)
- Coroutine (without heap allocation): `226 ns` (20x slower)

Performance

`view::concat`

Methodology: 3 `std::vector<int>`, 30 elements each, sum them

- Generator range: `11 ns`
- `views::concat`: `20 ns` (2x slower)
- Coroutine (without heap allocation): `226 ns` (20x slower)

`views::join`

Methodology: Nested `std::vector<std::vector<int>>`, each one with a different size, summing each element

- Generator range: `295 ns`
- `views::join`: `1067 ns` (3.6x slower)
- Coroutine (without heap allocation): `589 ns` (2x slower)

Performance

Compilers will learn to optimize coroutines better!

Performance

Compilers will learn to optimize coroutines better!

Maybe.

- What about debug performance?
- What about performance guarantees for time critical code?
- Can we recommend everybody to write input ranges using coroutines?

Performance

We're leaving room for a faster hand-written solution than ranges.

With generator ranges: We don't.

But: We need to support the range-based `for` loop, otherwise, nobody will use it.

Bonus: Ability to “iterate” a tuple

```cpp
template <typename ... T>
class tuple
{
public:
    auto operator()(auto&& sink) const
    {
        return [&]<std::size_t ... Idx>(std::index_sequence<Idx...>) {
            std::control_flow flow;
            // && short-circuits once we have a break-like control flow.
            ((flow = sink(std::get<Idx>(*this))) && ...);
            return flow;
        }(std::index_sequence_for<T...>{});
    }
};
```

Questions for EWG(I)

1. Do we agree that we need something that is as convenient as `std::generator` but with guaranteed better performance?
2. If so, do we like the direction of a range-based `for` loop whose body is turned into a callback?