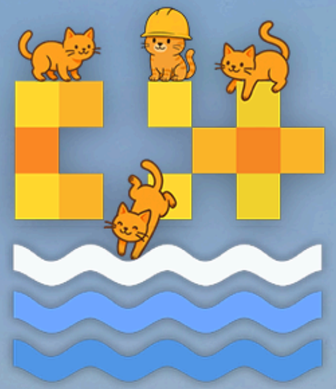

Understanding, Using, and Improving `std::generator`

Johannes Kalmbach

2025

About Myself (Johannes Kalmbach)

- C++ programmer since 2017
- PhD student at University of Freiburg (Germany)
- Co-founder of `QLeverize`
- First time speaker + attendee
- johannes.kalmbach@gmail.com
- https://github.com/joka921

`std::generator` - A Simple Example

```cpp
import std; // Save slide space.

std::generator<size_t> fibonacci_gen() {
    size_t i = 0, j = 1;
    while (true) {
        co_yield j;
        i = std::exchange(j, i + j);
    }
}

int main() {
    for (auto fib : fibonacci_gen() | std::views::take(10)) {
        std::println("yielded {}", fib);
    }
}
```

A `std::generator` generates a sequence of elements by repeatedly resuming the coroutine from which it was returned. Each time a `co_yield` statement is evaluated, the coroutine produces one element of the sequence.

C++ Standard, section 26.8.1

`std::generator` is...

- ... a convenient way to produce values one by one
- ... a `std::ranges::input_range`
- ... a coroutine mechanism (using C++20 `<coroutine>`s)
- ... a C++23 library type (can be implemented using C++20)
- ... inherently type-erased (like all coroutines)

`std::generator` is type-erased

```
1  // A coroutine of type `generator<size_t>`
2  generator<size_t> fibonacci_gen() {
3    /* as before, including `co_yield` */
4  }
5
6  // Another coroutine of type `generator<size_t>`
7  generator<size_t> fourty_two_gen() {
8    co_yield 42;
9  }
10
11 // An ordinary function that returns `generator<size_t>`
12 generator<size_t> choose_gen(bool b) {
13   return b ? fibonacci_gen() : forty_two_gen();
14 }
```

The Template Interface

```
1 template <typename Ref, typename V = void, typename Allocator = void>
2 class generator;
```

- Typical case: `Ref` specified, `V = void`
- Infer `reference_type` and `value_type` from `Ref`

Ref	reference_type	value_type
string	string&&	string
string&&	string&&	string
string&	string&	string
const string&	const string&	string

The Template Interface

```
1 template <typename Ref, typename V = void, typename Allocator = void>
2 class generator;
```

- If both `Ref` and `V` are specified, they are `reference_type` and `value_type` of the generator
- Needed for proxy reference types, e.g.

```
generator<tuple<const string&, const vector<int>&>, tuple<string, vector<int>>>
```

Caveat

```
1 std::generator<std::string> gen() {
2   std::string s;
3   // This line silently creates a copy.
4   co_yield s;
5 }
```

- `reference_type` is an rvalue-reference
- An `lvalue-reference` is `co_yielded`
- Then the yielded value is copied
- Mandated by the standard, currently not found on `cppreference.com`

Which of Those Have Issues?

```
1  generator<string> g1() {
2    std::string s;
3    co_yield s;
4  }
5
6  generator<string&&> g2() {
7    std::string s;
8    co_yield s;
9  }
10
11 generator<const string&> g3() {
12   std::string s;
13   co_yield s;
14 }
```

```
1  generator<string&> g4() {
2    const std::string s;
3    co_yield s;
4  }
5
6  generator<string&> g5() {
7    std::string s;
8    co_yield s;
9  }
```

Which of Those Have Issues?

```
1  generator<string> g1() {
2    std::string s;
3    // Silent copy
4    co_yield s;
5  }
6
7  generator<string&&> g2() {
8    std::string s;
9    // Silent copy
10   co_yield s;
11 }
12
13 generator<const string&> g3() {
14   std::string s;
15   co_yield s;
16 }
```

```
1  generator<string&> g4() {
2    const std::string s;
3    // Fails to compile.
4    co_yield s;
5  }
6
7  generator<string&> g5() {
8    std::string s;
9    co_yield s;
10 }
```

Custom Allocators

- Coroutines need to allocate
- Default is `std::allocator`

```
template<typename T>
class LoggingAllocator {
  constexpr T* allocate(size_t n) {
    std::println("Allocating {} bytes", n * sizeof(T));
    // Actual allocation...
  }

  constexpr void deallocate(T* p size_t n) { /* Analogously */}
  }
};

std::generator<int, void, LoggingAllocator<int>> gen() {
 co_yield  23;
 co_yield 64;
 };

int main() {
 { gen();}
 // Prints "Allocating 96 bytes" on GCC-14, and "...64 bytes" on Clang-19
}
```

std::ranges::elements_of

```cpp
// Example from `cppreference.com`
#include <generator>

template<typename T>
struct Tree
{
  T value;
  Tree *left{}, *right{};

  std::generator<const T&> traverse_inorder() const
  {
    if (left)
      co_yield std::ranges::elements_of(left->traverse_inorder());

    co_yield value;

    if (right)
      co_yield std::ranges::elements_of(right->traverse_inorder());
  }
};
```

- More efficient than `for (auto&& el : range) {co_yield std::foward<decltype(el)>(el);}`

Perfomance

Performance / Codegen

According to Trusted Community Members

> *Coroutines are simple and elegant, but the codegen is atrocious – definitely unsuitable for hot paths.*

Vittorio Romeo on his blog

> *... Well, that's relative ...*

Phil Nash on CppCast, starting at 16:00

> *... I mean, that's relative and it depends on your code and you have to measure and blablabla ...*

Timur Doumler, same place

... so let's measure ...

A Simple Benchmark

Compare various implementations of `fibonacci_gen`:

```cpp
// Generator, body visible in header
std::generator<size_t> fib_gen() { /* as before */}

// Can be completely inlined
struct Inlineable {
  size_t i = 0, j = 1;
  size_t next() {
        auto res = j;
        i = std::exchange(j, i + j);
        return res;
};

// Indirection via function call
struct NonInlineable {
  size_t i = 0, j = 1;
  size_t next(); // Defined in different TU
}

// Indirection via virtual function
struct RangeOfSizeTBase {
  virtual size_t next() = 0;
  virtual ~RangeOfSizeTBase = default;
}
std::unique_ptr<RangeOfSizeTBase> getVirtualFibRange();
```

Initial Benchmark Results

- Using `google::benchmark`
- AMD Ryzen 9 7950x
- time in nanoseconds/element

	GCC-14	Clang-19
`std::generator`	2.7	1.7
`inlineable class`	0.6	0.6
`non-inlineable class`	2.6	1.7
virtual function	2.6	1.7

Performance Analysis

- `std::generator` has to allocate
- Allocation implies indirection
- ... and hinders inlining
- The overhead introduced can be
  - Negligible / Acceptable
  - Significant / Inacceptable
  - Unavoidable
- HALO is allowed for coroutines...
- ... but neither GCC nor Clang currently apply it (Clang used to)
- Everything above holds for all type-erasure mechanisms.

Summary: When to Consider `std::generator`

- When you can afford the overhead
- When you need type-erasure anyway
  - See excellent talks by Klaus Iglberger about type erasure
- When you need to define a complex input range
  - Writing iterators manually is tedious
  - Where possible, consider `std::ranges`/`std::views`

Mitigating Performance Issues

Idea: Let the generator yield a batch of values at once

```
generator<const vector<size_t>&> batched_fib() {
  std::vector<size_t> buffer;
  buffer.reserve(BUF_SIZE);
  size_t i = 0, j = 1;
  while (true) {
    size_t next = j;
    i = std::exchange(j, i + j);
    buffer.push_back(next);
    if (buffer.size() == BUF_SIZE) {
      co_yield buffer;
      buffer.clear();
    }
  }
}
```

```
void consume_using_join_view() {
  for (size_t s : batched_iota()
                    | std::views::join) {
    // Do something.
  }
}

void consume_using_nested_loop() {
  for (const auto& batch : batched_iota()) {
    for (size_t s : batch) {
      // Do something
    }
  }
}
```

Updated Benchmark Results

	GCC-14	Clang-19
`std::generator`	2.7	1.7
`inlineable class`	0.6	0.6
`non-inlineable class`	2.6	1.7
virtual function	2.6	1.7
batched, `join_view`	1.1	1.1
batched, nested loop	1.1	1.1

Removing The Boilerplate, Version 1

- We can use a macro...
- ... or maybe reflection in the future?

```
#define CO_YIELD_BATCHED(buffer, el) \
    buffer.push_back(el); \
    if (buffer.size() == BUF_SIZE) { \
      co_yield buffer; \
      buffer.clear(); \
    } \
    void(0)

generator<const vector<size_t>&> batched_fib() {
  std::vector<size_t> buffer;
  buffer.reserve(BUF_SIZE);
  size_t i = 0, j = 1;
  while (true) {
    size_t next = j;
    i = std::exchange(j, i + j);
    CO_YIELD_BATCHED(buffer, i);
  }
}
```

Removing the Boilerplate, Version 2

- Previously: Manual batching of values (hidden by macro)
- Can we move the boilerplate into the `generator` type?

```cpp
#include "./batched_generator.hpp"

// Only choose a different return type, everything else stays the same.
// Second template argument is the batch size
batched_generator<size_t, 100> batched_fib() {
  size_t i = 0, j = 1;
  while (true) {
    // Yield values one by one
    size_t next = j;
    i = std::exchange(j, i + j);
    // Implicit batching by the machinery.
    co_yield next;
  }
}

int main() {
  for (size_t s : batched_fib()) {
    // Do something useful.
    std::println("{}", s);
  }
}
```

The promise_type (Simplified)

```cpp
// `Yielded` is the corresponding generators yielded type
// (e.g. `string&&` for `generator<string>`, or
//  `const string&` for `generator<const string&>`).
template<typename Yielded>
class PromiseType {
  // Store a pointer to the most recently yielded value
  add_pointer_t<Yielded> value_ = nullptr;

  // Called for each `co_yield` statement. Suspend the coroutine,
  // s.t. the value can be used by the `generator`.
  suspend_always yield_value(Yielded val) noexcept {
    value_ = std::addressof(val);
    return {};
  }

  /* Omitted: Special overload for `co_yield` that creates a copy if necessary */

  // Suspend the coroutine at the beginning and the end.
  suspend_always initial_suspend() const noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }

  /* Omitted: boilerplate code for exception handling and `co_return` */
};
```

The Batched promise_type (Simplified)

```cpp
template<typename T, size_t BatchSize = 100>
class BatchedPromiseType {
  static_assert(is_object_v<T>);

  std::vector<T> buffer_;
  // Called for each `co_yield` statement.
  // Suspend the coroutine only if the `buffer_` is full.
  template <typename U>
  SuspendIf yield_value(U&& val) noexcept {
    buffer_.emplace_back(std::forward<U>(val));
    return SuspendIf{buffer_.size() >= BatchSize};
  }

  // Has to be called when advancing the iterator
  void resetBuffer() {buffer_.clear(); buffer_.reserve(BatchSize)}

  // Suspend the coroutine at the beginning and the end.
  suspend_always initial_suspend() const noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }

  /* Omitted: boilerplate code for exception handling and `co_return` */
};
```

SuspendIf

- Consists purely of special functions that the `<coroutine>` framework expects

```
1   struct SuspendIf {
2     bool suspend_;
3
4     // If `true`, then the coroutine is NOT suspended
5     constexpr bool await_ready() noexcept { return !suspend_; }
6
7     // When suspending, return control to the caller
8     void await_suspend(std::coroutine_handle<>) noexcept {
9       return;
10    }
11    // Nothing to do when resuming the coroutine
12    constexpr void await_resume() const noexcept {}
13  };
```

The generator Class

```
1  template<typename Ref, typename Val>
2  class generator : public ranges::view_interface<generator<Ref, Val>> {
3    /* Omitted typedefs for `Value`, `Reference`, `Yielded`. */
4    using promise_type = PromiseType<Yielded>;
5
6  public:
7    /* Omitted: move operations (implemented) and copy operations (deleted) */
8
9    ~generator() { if (coro_) { coro_.destroy();}
10
11   // Iterator `resumes` the coroutine via the handle and accesses
12   // yielded values via the `promise`.
13   struct Iterator;
14   Iterator begin();
15   std::default_sentinel_t end() const noexcept;
16 private:
17   coroutine_handle<promise_type> coro_;
18
19   generator(coroutine_handle<promise_type> coro) noexcept
20           : coro_{move(coro)} {}
21 };
```

The batched_generator Class

```
template<typename T>
class batched_generator : public ranges::view_interface<generator<T> {
  using promise_type = BatchedPromiseType<T>;
public:
  /* Omitted: move operations, constructor, destructor (same as before) */

  struct Iterator;
  Iterator beginInternal() { return Iterator{coro_}; }

  void emplaceJoin() {
    if (!join_) {
       join_.emplace(std::views::join(stdr::subrange{beginInternal(), std::default_sentinel}));
    }
  }

  auto begin() {
    emplaceJoin();
    return join_->begin();
  }
  auto end() {
    emplaceJoin();
    return join_->end();
  }
private:
  std::optional<stdr::join_view<stdr::subrange<Iterator, std::default_sentinel_t>>> join_;
  coroutine_handle<promise_type> coro_;
};
```

The Valley of Tears

- `batched_generator` showed no performance improvements...
- Codegen of generator is very vulnerable to small changes
- Chicken vs. Egg

Disassembly of Hot Path, `iota_gen`

```
.L20:
    leaq    1(%rax), %rcx       ; i++
    movq    %rax, 80(%rbx)      ; Store old `i` to RAM
    movq    %rcx, 64(%rbx)      ; Store `i` to RAM
    cmpq    %rdx, %rbp          ; buf.size() < buf.capacity()
    je      .L10                ; Jump to reallocation
    movq    %rax, 0(%rbp)       ; push_back (*it = i)
    addq    $8, %rbp            ; it++
    movq    %rbp, 24(%rbx)      ; Store `it` to RAM
.L11:
    movq    %rbp, %rax
    subq    %r13, %rax
    cmpq    $792, %rax          ; buf.size() >= 100
    seta    %al
    movb    %al, 72(%rbx)       ; Store `SuspendIf::suspend`
    testb   %al, %al
    jne    .L21                 ; Jump to suspension
    movq    64(%rbx), %rax      ; Load `i` from RAM
    jmp    .L20                 ; Back to beginning of loop
```

Disassembly of Hot Path, `iota_gen`, Manually Improved

```
1  .L20:
2
3      movq    %rax, 0(%rbp)    ; push_back()
4      addq    $8, %rbp         ; ++it;
5
6      movq    %rbp, %rcx
7      subq    %r13, %rcx
8      cmpq    $792, %rcx       ; if (buf.size() >= 100
9      ja      .L21_pre         ; Jump to suspension
10
11     leaq    1(%rax), %rax    ; ++i;
12     jmp     .L20             ; Back to beginning of loop
13
14 /* dumping of registers before suspend ... */
```

- Performance as expected/desired
- Compilers need to learn this + HALO

Final Thoughts

- `std::generator` is useful and convenient
- `std::generator` has its performance issues
  - Be nice to your compiler vendors, maybe we get consistent HALO one day
  - ... and better codegen for the coroutine state machines
- `(std::)generator` is a good starting point for your `<coroutine>` journey
  - Available in the standard library
  - Not too complex (understandable for mere mortals)

End