++it Italian C++ Community

10 Years

www.italiancpp.org

Hello, std::generator

Alberto Barbati

Italian C++ Conference 2023
June 10, Rome

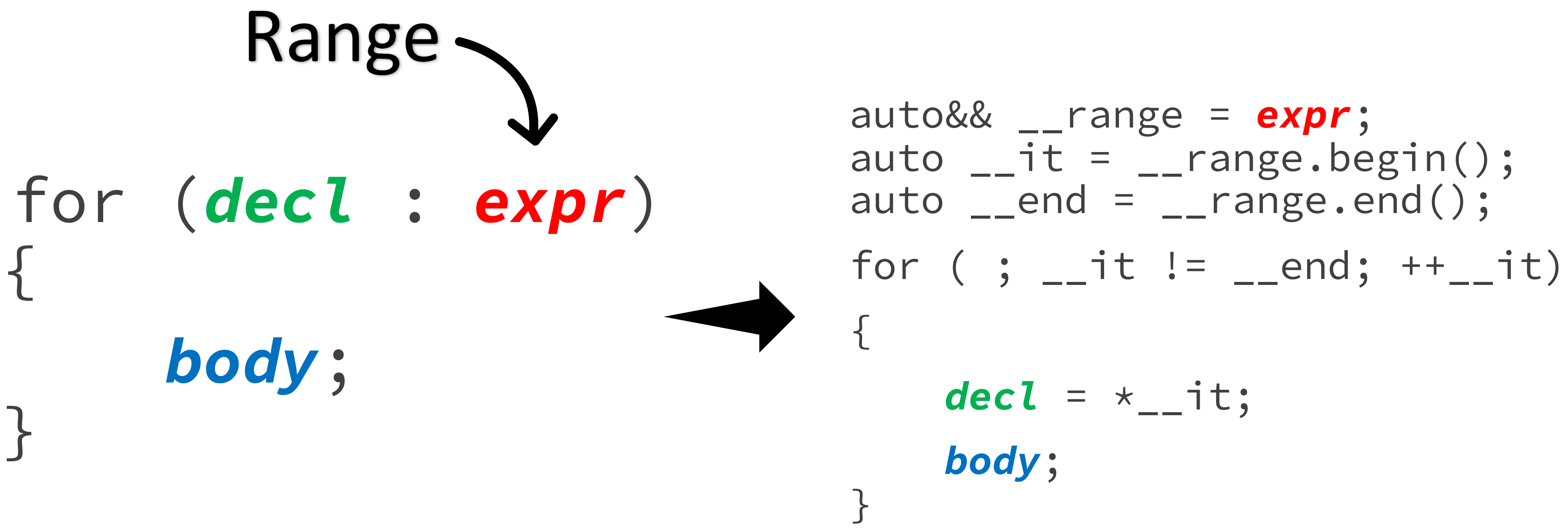
Plan for this session

We’ll introduce std::generator

- A new library facility coming with C++23
- Motivation and main use cases
- Comparison with traditional approaches

A reference implementation is available for C++20

Ranges for dummies



Range

```
for (decl : expr)
{
    body;
}
```

```
auto&& __range = expr;
auto __it = __range.begin();
auto __end = __range.end();

for ( ; __it != __end; ++__it)
{
    decl = *__it;
    body;
}
```

Motivating example

Generating Fibonacci numbers

```
std::vector<int> fibonacci(int n)
{
   std::vector<int> result;
   int a = 0, b = 1;
   while (n--)
   {
       result.push_back(b);
       int c = a + b;
       a = b; b = c;
   }
   return result;
}
```

Usage scenarios

```
// iterating

for (int x : fibonacci(10))
{
    std::print("{}, ", x);
}

// output: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,


// printing a range in C++23 is easier

std::println("{}", fibonacci(10));

// output: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Ranges composition

```
auto is_even = [](int n){ return n % 2 == 0; }


using std::views::filter;

std::println("{}", fibonacci(10) | filter(is_even));

// output: [2, 8, 34]


// filtered iteration is ok in C++23

for (int x : fibonacci(10) | filter(is_even))
{
    // use x
}
```

Stopping sooner

```
auto less_than = [](int max)
{
    return [max](int n) { return n < max; };
};

using std::views::take_while;


std::println("{}", fibonacci(10) | take_while(less_than(20)));

// output: [1, 1, 2, 3, 5, 8, 13]
```

Alternative #1: foreach-like

```cpp
void fibonacci(int n, std::invocable<int> auto out)
{
    int a = 0, b = 1;
    while (n--)
    {
        std::invoke(out, b);
        int c = a + b;
        a = b; b = c;
    }
}
```

Usage scenarios

```
// iterating

for (int x : fibonacci(10))
{
  std::print("{}, ", x);
}


// printing a range

std::println("{}", fibonacci(10));
```

```
// iterating

fibonacci(10, [](int x)
  {
     std::print("{}, ", x);
  });
```

😑

Alternative #2: writing a range

You need to:

- Provide a class with a begin() and end() function
- Provide classes for iterator and sentinel
- Put the generator state somewhere (either in the iterator or in the range object itself)
- Put the computation code in the iterator operator++
- Implement iterator/sentinel comparisons
- …

#include <generator>

Generator example

```
std::generator<int> fibonacci(int n)
{
   int a = 0, b = 1;
   while (n--)
   {
       co_yield b;
       int c = a + b;
       a = b; b = c;
   }
}
```

Flow of execution

```
for (int x : fibonacci(10))
{
    std::print("{}, ", *x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;

        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;

        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Flow of execution

```
auto&& __range = fibonacci(10);
auto __it = __range.begin();
auto __end = __range.end();
for ( ; __it != __end; ++__it)
{
    int x = *__it;
    print("{}, ", x);
}
```

```
std::generator<int> fibonacci(int n)
{
    int a = 0, b = 1;
    while (n--)
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

And so on…

Usage scenarios

```
// iterating
for (int x : fibonacci(10))
{
    std::print("{}, ", x);
}
// output: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

😇

```
// printing a range in C++23 is easier
std::println("{}", fibonacci(10));
// output: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

😇

Composition with ranges

```
auto is_even = [](int n){ return n % 2 == 0; }
```

```
using std::views::filter;

std::println("{}", fibonacci(10) | filter(is_even));

// output: [2, 8, 34]
```

😇

```
// filtered iteration is ok in C++23

for (int x : fibonacci(10) | filter(is_even))
{
    // use x
}
```

😇

Stopping sooner

```
using std::views::take_while;

auto less_than = [](int max)
{
    return [max](int n) { return n < max; };
};

std::println("{}", fibonacci(10) | take_while(less_than(20)));

// output: [1, 1, 2, 3, 5, 8, 13]
```

Endless generator

```
std::generator<int> fibonacci()
{
    int a = 0, b = 1;
    for (;;) // infinite loop
    {
        co_yield b;
        int c = a + b;
        a = b; b = c;
    }
}
```

Endless generator usage

```
using namespace std::views;

std::println("{}", fibonacci() | take(10));

// output: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

std::println("{}", fibonacci() | take_while(less_than(20)));

// output: [1, 1, 2, 3, 5, 8, 13]
```

Nested generators

Visiting a tree structure

```
struct node
{
  node* left;
  node* right;
  int payload;
};
```

Classic visit approach

```
void visit( node* root,
            std::invocable<int> auto out)
{
  if (root)
  {
      visit(root->left, out);
      std::invoke(out, root->payload);
      visit(root->right, out);
  }
}
```

Generator approach (naïve)

```
std::generator<int> visit(node* root)
{
   if (root)
   {
      for (int x : visit(root->left))
         co_yield x;

      co_yield root->payload;

      for (int x : visit(root->right))
         co_yield x;
   }
}
```

Generator approach (efficient)

```
std::generator<int> visit(node* root)
{
  using std::ranges::elements_of;
  if (root)
  {
      co_yield elements_of(visit(root->left))
      co_yield root->payload;
      co_yield elements_of(visit(root->right));
  }
}
```

Recap

A generator is a function that produces a sequence of values:

- One value at a time;
- Values are computed only when needed;
- Behaves as a C++20 range

Bonus: code is as easy to use and maintain wrt other approaches

Uses C++20 coroutines syntax

Fine prints and caveats

There is a little overhead, due to the memory allocation of the coroutine context

Generators are input ranges, so you can iterate them only once and only in the forward direction

Generators and their iterators are move-only types

The interface of `std::generator` is carefully designed to avoid unnecessary copies in the yield process

Advanced usages

The `std::generator` template has two "advanced" template parameters that allows you to:

- Improve interoperability with other ranges, when returning object types that actually play the role of references, such as string_view or span
- Customize the memory allocation of the coroutine context object: stateless, stateful and PMR allocators are all supported

References

Reference paper by Casey Carter

https://wg21.link/P2502R2

Reference implementation by Casey Carter, Lewis Baker and Corentin Jabot:

https://godbolt.org/z/5hcaPcfvP

PMR extension by Steve Downey:

https://wg21.link/P2787R0

Thanks for you attention

Questions?

Alberto Barbati
@gamecentric
alberto@gamecentric.com