Putting `integer_sequence` on a diet

Vittorio Romeo

vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

@supahvee1234

C++ Now 2019

2019/05/08

Aspen, CO

Bloomberg

- Iterate over a *variadic pack*, providing both the *index* and the *element*
  - The *index* must be usable in a *constant expression*

```
template <typename... Xs>
void print_with_index(const Xs&... xs)
{
    enumerate([](auto i, const auto& x)
    {
        static_assert(i < sizeof...(xs));
        std::cout << i << ": " << x << '\n';
    }, xs...);
}
```

```
print_with_index("hello", 42, 'x');
// 0: hello
// 1: 42
// 2: x
```

solution #0
indirection via non-member function

```
template <typename F, typename ... Xs>
void enumerate(F&& f, Xs&& ... xs)
{
    enumerate_impl(std::index_sequence_for<Xs ... >{},
                   std::forward<F>(f),
                   std::forward<Xs>(xs) ... );
}
```

```
template <std::size_t ... Is, typename F, typename ... Xs>
void enumerate_impl(std::index_sequence<Is ... >, F&& f, Xs&& ... xs)
{
    (f(std::integral_constant<std::size_t, Is>{},
       std::forward<Xs>(xs)), ... );
}
```

(on *wandbox.org*)

- Verbose, user syntax is annoying

solution #1

nicer user syntax with C++20 template lambdas

```
template <typename ... Xs>
void print_with_index(Xs ... xs)
{
    enumerate([]<auto I>(auto x)
    {
        std::cout << I << ": " << x << '\n';
    }, xs ... );
}
```

```
template <std::size_t ... Is, typename F, typename ... Xs>
void enumerate_impl(std::index_sequence<Is ... >, F&& f, Xs&& ... xs)
{
    (f.template operator()<Is>(std::forward<Xs>(xs)), ... );
}
```

(on *wandbox.org*)

- Nicer - can we use this to avoid `enumerate_impl`?

solution #2

`enumerate` with C++20 template lambdas

```
template <typename F, typename... Xs>
void enumerate(F&& f, Xs&&... xs)
{
    [&]<auto... Is>(std::index_sequence<Is...>)
    {
        (f.template operator()<Is>(std::forward<Xs>(xs)), ...);
    }(std::index_sequence_for<Xs...>{});
}
```

(on *wandbox.org*)

- Avoid indirection with `enumerate_impl`
- Less boilerplate, no need to pass `f` and `xs...` again

- **P1306**: "*Expansion statements*" (A. Sutton, S. Goodrick, D. Vandevoorde)
  - Approved by EWG (Kona 2019)

```
for ... (auto elem : std::tuple{0, â€˜aâ€™, 3.14})
    std::cout << elem << '\n';
```

```
std::cout << std::get<0>(t) << '\n';
std::cout << std::get<1>(t) << '\n';
std::cout << std::get<2>(t) << '\n';
```

- Supports "iteration" over entities that bind to *structured bindings*
  - (among others)


```
for ... (auto i : std::make_index_sequence<3>{})
    std::cout << i << '\n';
```

- Does this work?

```
auto [a, b, c, d] = std::make_index_sequence<3>{};
```

(on *godbolt.org*)

```
error: cannot decompose class type 'std::integer_sequence<...>' [...]
   5 |     auto [a, b, c, d] = std::make_index_sequence<3>{};
     |           ^~~~~~~~~~~~
```

- No, it doesn't
  - Can we make it work?

can we do better?
expansion statement

```
template <auto ... Is>
struct sequence { };

template <auto I, auto ... Is>
constexpr auto get(sequence<Is ... >) { return std::array{Is ... }[I]; }
```

```
template <auto ... Is>
struct tuple_size<my::sequence<Is ... >>
    : std::integral_constant<std::size_t, sizeof ... (Is)> { };

template <std::size_t I, auto ... Is>
struct tuple_element<I, my::sequence<Is ... >>
    : std::type_identity<std::common_type_t<decltype(Is) ... >> { };
```

```
auto [a, b, c, d] = sequence<0, 1, 2, 3>{};
```

(on *godbolt.org*)

- Working with **Alisdair Meredith** to make `integer_sequence` decomposable
  - The following will very likely be possible in C++20/23

```
template <typename F, typename ... Xs>
void enumerate(F&& f, Xs&& ... xs)
{
    auto t = std::forward_as_tuple(std::forward<Xs>(xs) ... );
    for ... (auto I : std::index_sequence_for<Xs ... >{})
    {
        f.template operator()<I>()(std::get<I>(t));
    }
}
```


- **P1061**: "*Structured Bindings can introduce a Pack*" (B. Revzin, J. Wakely)
  - EWGI wants to see it again (Kona 2019)

```
template <typename F, typename ... Xs>
void enumerate(F&& f, Xs&& ... xs)
{
    constexpr auto [ ... Is] = std::index_sequence_for<Xs ... >{};
    (f.template operator()<Is>(std::forward<Xs>(xs)), ... );
}
```

- My favorite solution
  - Less confident that this will be available in C++20/23 (or ever)

Thanks!

https://vittorioromeo.info

https://github.com/SuperV1234/cppnow2019

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

@supahvee1234

Bloomberg

