Introduction to C++ origami

Bloomberg

Vittorio Romeo
https://vittorioromeo.info
vittorio.romeo@outlook.com

C++::London
10/01/2018

About me

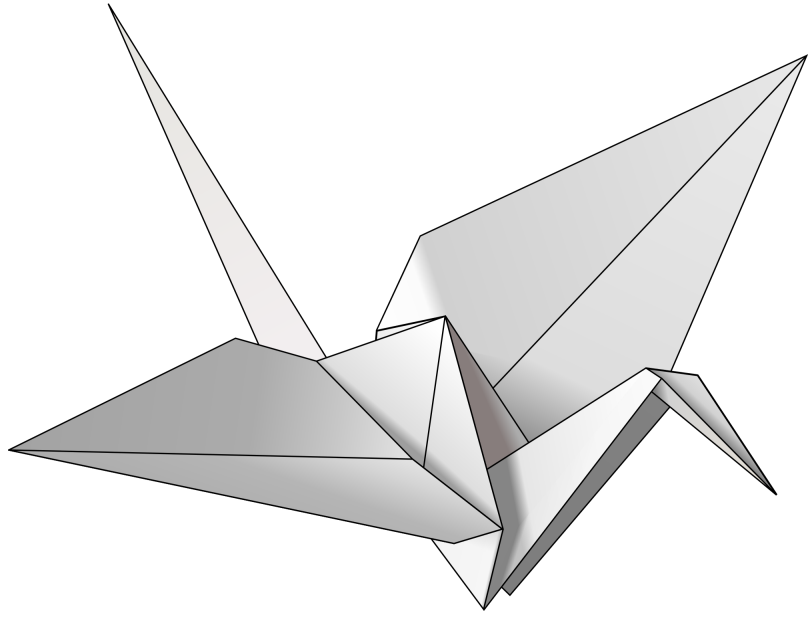
- Developer at **Bloomberg L.P.**
- Modern C++ enthusiast
  - Conference talks **(YouTube playlist)** | **(SkillsMatter)**
  - Video tutorials & articles **(vittorioromeo.info)**
  - Open-source projects **(GitHub/SuperV1234)**

Slides

- **https://github.com/SuperV1234/cpplondon**

This talk is about

fold expressions



- Fold expressions allow us to **reduce** a *parameter pack* over a *binary operator*
- They were added in C++17
- They concisely generate code that:
  - Performs an action on every element of a *parameter pack*
  - Reduces elements of a *parameter pack* in a single final result

Printing a *parameter pack* in C++11 - recursive approach

```
template <typename X>
void print(const X& x)
{
    std::cout << x;
}

template <typename X, typename... Xs>
void print(const X& x, const Xs&... xs)
{
    print(x);
    print(xs...);
}
```

on **wandbox.org**

Printing a *parameter pack* in C++11 - direct approach

```
template <typename... Xs>
void print(const Xs&... xs)
{
    (void) std::initializer_list<int>{
        ((std::cout << xs), 0)...
    };
}
```

on **wandbox.org**

Printing a *parameter pack* in C++14 - direct approach

```
template <typename ... Xs>
void print(const Xs& ... xs)
{
    for_each_argument([](const auto& x)
    {
        std::cout << x;
    }, xs ... );
}
```

If interested, check out my talk from CppCon 2015:

" `for_each_argument` explained and expanded"

Printing a *parameter pack* in C++17 - fold expression

```
template <typename ... Xs>
void print(const Xs& ... xs)
{
    (std::cout << ... << xs);
}
```

on **wandbox.org**

fold expression(since C++17)

Reduces (folds) a parameter pack over a binary operator.

Syntax

(*pack op* ...)	(1)
(... *op pack*)	(2)
(*pack op* ... *op init*)	(3)
(*init op* ... *op pack*)	(4)

1) unary right fold
2) unary left fold
3) binary right fold
4) binary left fold

op - any of the following 32 *binary* operators: `+` `-` `*` `/` `%` `^` `&` `|` `=` `<` `>` `<<` `>>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<=` `>>=` `==` `!=` `<=` `>=` `&&` `||` `,` `.*` `->*`. In a binary fold, both *op*s must be the same.

pack - an expression that contains an unexpanded parameter pack and does not contain an operator with precedence lower than cast at the top level (formally, a *cast-expression*)

init - an expression that does not contain an unexpanded parameter pack and does not contain an operator with precedence lower than cast at the top level (formally, a *cast-expression*)

Note that the open and closing parentheses are part of the fold expression.

from **http://en.cppreference.com/w/cpp/language/fold**

```cpp
template <typename ... Xs>
void print(const Xs& ... xs)
{
//                op          pack
//                v~          v~
    (std::cout << ... << xs);
//   ^~~~~~~~~~          ^~
//   init                op
}
```

This is a **binary left fold**.

Explanation

The instantiation of a *fold expression* expands the expression e as follows:

1) Unary right fold $(E\ op\ ...)$ becomes $E_1\ op\ (...\ op\ (E_{N-1}\ op\ E_N))$

2) Unary left fold $(...\ op\ E)$ becomes $((E_1\ op\ E_2)\ op\ ...)\ op\ E_N$

3) Binary right fold $(E\ op\ ...\ op\ I)$ becomes $E_1\ op\ (...\ op\ (E_{N-1}\ op\ (E_N\ op\ I)))$

4) Binary left fold $(I\ op\ ...\ op\ E)$ becomes $(((I\ op\ E_1)\ op\ E_2)\ op\ ...)\ op\ E_N$

(where N is the number of elements in the pack expansion)

```cpp
template <typename... Xs>
void print(const Xs&... xs)
{
    (std::cout << ... << xs);
}

print(1, 'a', 2);
```

> 4) Binary left fold $(I \ op \ ... \ op \ E)$ becomes $(((I \ op \ E_1) \ op \ E_2) \ op \ ...) \ op \ E_N$

```cpp
((std::cout << 1) << 'a') << 2
```

```cpp
template <typename T, typename ... Xs>
void push_back(std::vector<T>& v, Xs&& ... xs)
{
    (v.push_back(std::move(xs)), ... );
}

push_back(v, 1, 2, 3);
```

> 1) Unary right fold $(E\ op\ ...)$ becomes $E_1\ op\ (...\ op\ (E_{N-1}\ op\ E_N))$

```cpp
v.push_back(1), (v.push_back(2), v.push_back(3))
```

Note that precedence/associativity and sequencing order is given by the chosen **operator**, not by the parenthesis!

C++ Operator Precedence

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	`type``()` *`type`*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
...	...	...	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Compound assignment by sum and difference	
	`*= /= %=`	Compound assignment by product, quotient, and remainder	
	`<<= >>=`	Compound assignment by bitwise left shift and right shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

from http://en.cppreference.com/w/cpp/language/operator_precedence

9) Every value computation and side effect of the first (left) argument of the built-in comma operator `,` is *sequenced before* every value computation and side effect of the second (right) argument.

from **http://en.cppreference.com/w/cpp/language/eval_order**

Some *cool* things you can do with

fold expressions

Comma-separated print

```
template <typename X, typename... Xs>
void cs_print(const X& x, const Xs&... xs)
{
    std::cout << x;
    ((std::cout << ", " << xs), ...);
}
```

on **wandbox.org**

```
cs_print(1, 2, 3, 'a', 'b', 'c');
```

> 1, 2, 3, a, b, c

Concatenate objects into `std::string`

```
template <typename ... Xs>
std::string cat(Xs&& ... xs)
{
    std::ostringstream oss;
    (oss << ... << xs);
    return oss.str();
}
```

on **wandbox.org**

```
std::cout << cat("meow", "purr") << '\n';
```

> meowpurr

Checking if x is any of $\{xs...\}$

```
if(foo == 'a' || foo == 'c' || foo == 'e')
{
    // ... do something ...
}
```

- `foo ==` is repeated multiple times

```cpp
template <typename X, typename... Xs>
constexpr bool is_any_of(const X& x, const X&... xs)
{
    return ((x == xs) || ...);
}
```

```cpp
if(is_any_of(foo, 'a', 'c', 'e'))
{
    // ...do something...
}
```

on **wandbox.org**

...with some additional helpers:

```
if(any_of('a', 'b', 'c').is(foo))
{
    // ... do something ...
}
```

on **wandbox.org**

Iteration from 0 to N at compile-time

```
repeat<32>([](auto i)
{
    std::array<int, i> arr;
    // ... use `arr` ...
});
```

- `i` is an `std::integral_constant`
- The closure is invoked 32 times

```
template <auto N, typename F>
void repeat(F&& f)
{
    repeat_impl(f, std::make_index_sequence<N>{});
}
```

- `N` is explicitly provided by the user
- `F` is deduced
- `std::make_index_sequence` creates a compile-time integer sequence from 0 to N *(non-inclusive)*

```cpp
template <typename F, auto ... Is>
void repeat_impl(F&& f, std::index_sequence<Is ... >)
{
    (f(std::integral_constant<std::size_t, Is>{}), ... );
}
```

- "Match" the generated sequence into `Is ...`
- Invoke `f` N times using a *fold expression* over the *comma operator*

```cpp
template <typename F, auto ... Is>
void repeat_impl(F&& f, std::index_sequence<Is ... >)
{
    (f(std::integral_constant<std::size_t, Is>{}), ... );
}

template <auto N, typename F>
void repeat(F&& f)
{
    repeat_impl(f, std::make_index_sequence<N>{});
}
```

on **wandbox.org**

- "abstraction design and implementation: `repeat`"
- "compile-time `repeat` & `noexcept`-correctness"

Looping over the elements of `std::tuple`

```
template <typename F, typename Tuple>
void for_tuple(F&& f, Tuple&& tuple)
{
    std::apply([&f](auto&& ... xs)
    {
        (f(std::forward<decltype(xs)>(xs)), ... );
    }, std::forward<Tuple>(tuple));
}
```

- `std::apply` invokes a function by "unpacking" all the elements of a tuple as arguments
- The provided function uses a *fold expression* over the *comma operator* to invoke `f` for each tuple element

```
for_tuple([](const auto& x)
{
    std::cout << x;
}, std::tuple{1, 2, 'a', 'b'});
```

on **wandbox.org**

> 12ab

Looping over a set of types

```
for_types<int, float, char>([](auto t)
{
    using type = typename decltype(t)::type;
    // ...use `type`...
});
```

- The passed closure is invoked for each type
- `t` is an empty object carrying information about the current type

```
template <typename T>
struct type_wrapper
{
    using type = T;
};
```

- `type_wrapper` stores information about a type inside an empty object that can be used like a value
- It will be passed to the user-provided lambda
- "Type-value encoding" idiom

```
template <typename ... Ts, typename F>
void for_types(F&& f)
{
    (f(type_wrapper<Ts>{}), ...);
}
```

- `Ts ...` are explicitly provided by the user
- `F` is deduced
- A *fold expression* over the *comma operator* invokes `f` with every type

```
struct A { void foo() { std::cout << "A\n"; } };
struct B { void foo() { std::cout << "B\n"; } };
struct C { void foo() { std::cout << "C\n"; } };

for_types<A, B, C>([](auto t)
{
    using type = typename decltype(t)::type;
    type{}.foo();
});
```

on **wandbox.org**

> A
>
> B
>
> C

Check typelist uniqueness

```
template <typename ... >
inline constexpr auto is_unique = std::true_type{};

template <typename T, typename ...  Rest>
inline constexpr auto is_unique<T, Rest ... > =
    std::bool_constant<(!std::is_same_v<T, Rest> &&  ... )
                    && is_unique<Rest ... >>{};
```

- C++14 *variable templates* can be specialized
- Variables can be `inline` since C++17
- `std::bool_constant<X>` was introduced in C++17 - it's an alias for `std::integral_constant<bool, X>`

Base case

```
template <typename ... >
inline constexpr auto is_unique = std::true_type{};
```

- An empty type list is unique

Recursive case

```
template <typename T, typename ... Rest>
inline constexpr auto is_unique<T, Rest ... > =
    std::bool_constant<(!std::is_same_v<T, Rest> && ... )
                    && is_unique<Rest ... >>{};
```

- `<T, Rest ... >` type is unique if:
  - `Rest ...` does **not** contain `T`
  - `<Rest ... >` is an unique type list
- The "contains" check uses a *fold expression* over the `&&` operator

```
static_assert(is_unique<>);
static_assert(is_unique<int>);
static_assert(is_unique<int, float, double>);
static_assert(!is_unique<int, float, double, int>);
static_assert(!is_unique<int, float, double, int,
                         char, char>);
static_assert(is_unique<int, float, double, char>);
```

on **wandbox.org**

Short-circuiting JSON visitation

```
const auto& type = some_json_object["type"];

     if(type == "foo") { handle_foo(some_json_object); }
else if(type == "bar") { handle_bar(some_json_object); }
else if(type == "baz") { handle_baz(some_json_object); }
```

```
json_switch(some_json_object["type"],
    on{"foo"} | handle_foo,
    on{"bar"} | handle_bar,
    on{"baz"} | handle_baz
);
```

```
on{"foo"} | handle_foo
```

```
struct on
{
    const char* _key;
    constexpr on(const char* key) : _key{key} { }
};
```

```
template <typename F>
constexpr auto operator|(const on& o, F&& f)
{
    return handler{o._key, std::move(f)};
}
```

- `on` is used for `operator|` overloading

```
template <typename F>
struct handler : F
{
    const char* _key;
    constexpr handler(const char* key, F&& f)
        : F{std::move(f)}, _key{key}
    {
    }
};
```

- Binds a `const char*` key to a function

```cpp
template <typename ... Handlers>
void json_switch(const json& j, Handlers&& ... hs)
{
    (void)
    (
        (j.key() == hs._key
            ? (hs(j), true) : false) || ...
//                  ^~~~~
//                  short-circuiting
    );
}
```

on **wandbox.org**

More interesting snippets/discussions:

- "Syntactic sugar Sunday: `any_of` & `all_of`": https://twitter.com/supahvee1234/status/937116720195670016
- "Avoid if-else branching in string to type dispatching": https://stackoverflow.com/questions/48025783/avoid-if-else-branching-in-string-to-type-dispatching

Useful resources

- "Fun with folds" https://ngathanasiou.wordpress.com/2015/12/15/182/
- "Lazily evaluated folds in C++" https://ngathanasiou.wordpress.com/2016/03/22/lazily-evaluated-folds-in-c/

Thanks!

- https://vittorioromeo.info
- vittorio.romeo@outlook.com
- vromeo5@bloomberg.net
- @supahvee1234

https://github.com/SuperV1234/cpplondon