C++17 Metaprogramming

Section 6

In this section

- `if constexpr ( ... )`
- Fold expressions

if constexpr (...)

Part 6.1

In the past...

- Branching at compile-time was often cumbersome
- Regular `if` statements are not powerful enough

```
template <typename Component>
void registry::add_component(Component& component)
{
    if (has_initialize_v<Component>)
    {
        component.initialize();
    }

    track_component(component);
}
```

In the past...

```
template <typename Component>
void registry::add_component(Component& component)
{
    if (has_initialize_v<Component>) { component.initialize(); }
    track_component(component);
}
```

```
struct test_component { };

test_component tc;
registry r;
r.add_component(tc); // Compile-time error
```

```
error: 'test_component' has no member named 'initialize'
```

In the past...

- Even if the condition given to a regular `if` is a *constant expression*, all branches are instantiated
- A possible workaround is using *tag dispatch*
    - Verbose and cumbersome

```
template <typename Component>
void registry::try_to_initialize(std::true_type, Component&);

template <typename Component>
void registry::try_to_initialize(std::false_type, Component&);
```

```
try_to_initialize(has_initialize<Component>{}, component);
```

In C++17...

- C++17 introduces a new construct, `if constexpr`

```
if constexpr (/* condition */)
{
    // `true` branch
}
else
{
    // `false` branch
}
```

  - The provided `condition` **must** be a *constant expression*
  - Only the taken branch is instantiated, the other one isn't

In C++17...

```
template <typename Component>
void registry::add_component(Component& component)
{
    if constexpr (has_initialize_v<Component>) { component.initialize(); }
//      ^~~~~~~~~
    track_component(component);
}
```

```
struct test_component { };
struct init_component { void initialize(); };

test_component tc;
init_component ic;

registry r;
r.add_component(tc); // OK
r.add_component(ic); // OK, calls `ic.initialize()`
```

`if constexpr` - recursion

- `if constexpr` works well to control compile-time recursion

```
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    std::cout << x;
    if constexpr (sizeof...(Ts) == 0)
    {
        std::cout << '\n';
    }
    else
    {
        std::cout << ' ';
        print_with_spaces(xs...);
    }
}
```

(on godbolt.org)

`if constexpr` - specialization

- `if constexpr` makes it easy to specialize algorithms

```
template <typename T>
constexpr bool fuzzy_equality(const T& x, const T& y)
{
    if constexpr (std::is_floating_point_v<T>)
    {
        return std::abs(x - y) < T(0.0001);
    }
    else
    {
        return a == b;
    }
}
```

`if constexpr` - closing thoughts

- Compared to *overloading* or *template specialization*, `if constexpr`...
  - ...is more readable and requires less boilerplate;
  - ...is faster at compile-time;
  - ...is more "closed", users cannot add new branches.
- There is no `constexpr` *ternary operator*

Fold expressions

Part 6.2

In the past...

- Dealing with variadic *parameter packs* prior to C++17 is not easy
  - How to generate code for each element in the pack?
  - How to collapse the pack into a final single result?
- Some techniques can be used
  - Recursion
  - Arbitrary expansion context *(e.g.* `std::initializer_list`*)*

Adding elements together - C++11 with recursion

```
template <typename T>
auto add(const T& x)
{
    return x;
}

template <typename T, typename ... Ts>
auto add(const T& x, const Ts& ... xs)
{
    return x + add(xs ... );
}
```

(on *godbolt.org*)

- Requires two overloads
- Slow to compile

Adding elements together - C++11 with `std::intializer_list`

```
template <typename T, typename... Ts>
auto add(const T& x, const Ts&... xs)
{
    std::common_type_t<T, Ts...> acc{x};
    (void) std::initializer_list<bool>{
        ((acc += xs), true)...
    };

    return acc;
}
```

(on godbolt.org)

- Arcane technique, hard to read and to explain
- Requires state and mutability
- Faster to compile

In C++17...

- C++17 introduces *fold expressions*
  - They "collapse" a *parameter pack* into a single result, using a specified binary operator

```
template <typename... Ts>
auto add(const Ts&... xs)
{
    return (xs + ...);
}
```

(on *godbolt.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.

Explanation

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

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

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

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

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

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

Fold expressions - example

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

- The above 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)

Fold expressions - example

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

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

(on godbolt.org)

↓

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

Fold expressions - ordering

- Precedence, associativity, and sequencing order are 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

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.

Fold expression - comma operator example

```
template <typename Vector, typename... Ts>
void push_back_all(Vector& vec, Ts&&... xs)
{
    (vec.push_back(std::forward<Ts>(xs)), ...);
}
```

```
push_back_all(vec, 1, 2, 3);
```

↓

```
vec.push_back(1), (vec.push_back(2), vec.push_back(3));
// {1, 2, 3}
```

Fold expression - comma operator example

```
template <typename Vector, typename... Ts>
void push_back_all(Vector& vec, Ts&&... xs)
{
    (..., vec.push_back(std::forward<Ts>(xs)));
}
```

```
push_back_all(vec, 1, 2, 3);
```

↓

```
(vec.push_back(1), vec.push_back(2)), vec.push_back(3);
// {1, 2, 3}
```

Fold expression - improving `print_with_spaces`

- From the previous section

```cpp
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    std::cout << x;
    if constexpr (sizeof...(Ts) == 0)
    {
        std::cout << '\n';
    }
    else
    {
        std::cout << ' ';
        print_with_spaces(xs...);
    }
}
```

Fold expression - improving `print_with_spaces`

```
template <typename T, typename... Ts>
void print_with_spaces(const T& x, const Ts&... xs)
{
    std::cout << x;
    ((std::cout << ' ' << xs), ...);
    std::cout << '\n';
}
```

- No recursion
  - Simpler
  - Faster to compile
  - Easier to read

Fold expression - use cases

- Fold expressions are useful whenever you need to...
  - ...generate code for each element in a parameter pack;
  - ...collapse a parameter pack into a single result.
- In practice, this translates to:
  - Helper functions that reduce boilerplate
  - Avoidance of recursion and speeding up compilation time

Fold expression - concatenation

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

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

> meowpurr

Fold expression - repeated comparisons

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

- `foo ==` is repeated multiple times

Fold expression - repeated comparisons

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

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

Fold expression - repeated comparisons

- Syntax can be improved with a helper class

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

Fold expression - compile-time unrolling

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

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

Fold expression - compile-time unrolling

```cpp
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)*

Fold expression - compile-time unrolling

```
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*

Fold expression - compile-time unrolling

```
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*)

Fold expression - iteration over `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

Fold expression - iteration over `std::tuple`

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

(on *wandbox.org*)

> 12ab

Fold expression - iteration 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

Fold expression - iteration over a set of types

```cpp
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

Fold expression - iteration over a set of types

```
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

Fold expression - iteration over a set of types

```cpp
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

Fold expression - 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>`

Fold expression - check typelist uniqueness

Base case

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

- An empty type list is unique

Fold expression - check typelist uniqueness

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

Fold expression - check typelist uniqueness

```cpp
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*)

Section recap

- `if constexpr` greatly simplifies branching at compile-time
  - Supersedes template trickery in most cases
  - Not powerful enough in others (e.g. generating data members)
- Fold expressions provide a clean way of reducing parameter packs
  - Useful to repeat an action for every element of a pack…
  - …or to collapse a pack into a single result

Discussion

Use cases for metaprogramming in your projects

Exercise

- Implement compile-time loops with fold expressions
  - `exercise4.cpp`
    - on Wandbox
    - on Godbolt

Q&A

Break

5 minutes