Variadic Templates and Parameter Packs

Vladimir Vishnevskii

2025

Variadic templates

- Essential for modern C++ design
- Enable various standard and 3rd-party components: std::tuple, std::variant, std(fmt)::format e.t.c
- Can also be used to develop custom efficient and convenient facilities with limited amount of metaprogramming

In the talk

- "C++ fundamentals" talk: focuses on basic concepts with broad but shallow overview of potential applications
- Talk is not targeted at any specific (starting from C++11) standard: common techniques are demonstrated

Class and function templates

- Class (structure) template:

```
template<typename T>
class Foo {};
```

- Function template:

```
template<typename Arg>
void foo(Arg arg) {
    // ...
}
```

Class and function templates

- Class template:

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

Class and function templates

- Class template:

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

T is a type template parameter

Class and function templates

- Class template:

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

`Capacity` is a non-type template parameter

Template instantiation

- Instantiated with arguments:

```
static_vector<int, 10> v;
```

- Class after substitution (pseudocode):

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

```
class static_vector<T=int, Capacity=10> {
public:
    int& operator[](size_t) {/*...*/}
    //...

private:
    int storage_[10];
    //...
};
```

Template instantiation

- Instantiated with arguments:

```
static_vector<int, 10> v;
```

- Class after substitution (pseudocode):

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

```
class static_vector<T=int, Capacity=10> {
public:
    int& operator[](size_t) {/*...*/}
    //...

private:
    int storage_[10];
    //...
};
```

Instantiation produces concrete class

Template instantiation

- Instantiated with arguments:

```
static_vector<int, 10> v;
```

- Class after substitution (pseudocode):

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

```
class static_vector<T=int, Capacity=10> {
public:
    int& operator[](size_t) {/*...*/}
    //...

private:
    int storage_[10];
    //...
};
```

T is substituted by `int`

Template instantiation

- Instantiated with arguments:

```
static_vector<int, 10> v;
```

- Class after substitution (pseudocode):

```
template <typename T, size_t Capacity>
class static_vector {
public:
    T& operator[](size_t) {/*...*/}
    //...

private:
    T storage_[Capacity];
    //...
};
```

```
class static_vector<T=int, Capacity=10> {
public:
    int& operator[](size_t) {/*...*/}
    //...

private:
    int storage_[10];
    //...
};
```

`Capacity` is substituted by compile time constant 10

Variadic templates

- Class (struct) template:

```
template<typename... Ts>
class Foo {};
```

- Function template:

```
template<typename... Ts>
auto foo(Ts... args) -> void {
    // ...
}
```

Variadic templates

- Class (struct) template

```
template<typename... Ts>
class Foo {};
```

Ts is a template parameter pack

- Function template:

```
template<typename... Ts>
auto foo(Ts... args) -> void {
    // ...
}
```

Variadic templates

- Class (struct) template:

```
template<typename... Ts>
class Foo {};
```

- Function template:

```
template<typename... Ts>
auto foo(Ts... args) -> void {
    // ...
}
```

`args` is a function parameter pack

Parameter packs

- Parameter pack is a named entity that represents a group of arguments (types or values) used to instantiate variadic template
- Pack can have 0 or more items
- Inside a template's body packs should be explicitly "expanded"
- Parameter packs can be expanded within specific contexts and their usage will depend on the context in which they are expanded
- Comprehensive description of various contexts is available in *cppreference*

Parameter packs. Expansion contexts

- Template argument lists, function parameter list and brace(parentheses) - enclosed initializers:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

Parameter packs. Expansion contexts

- Template argument lists, function parameter list and brace(parentheses) - enclosed initializers:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

Template parameter pack

Parameter packs. Expansion contexts

- Template argument lists, function parameter list and brace(parentheses) - enclosed initializers:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

Expanded into template argument list

Parameter packs. Expansion contexts

- Template argument lists, function parameter list and brace(parentheses) - enclosed initializers:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

Into function parameter list

Parameter packs. Expansion contexts

- Template argument lists, function parameter list and brace(parentheses) - enclosed initializers:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

Initializer (can be with braces or parentheses)

Parameter packs. Expansion contexts

- Instantiation:

```
tuple_wrapper<int, float> w{1, 2.0};
```

- Substitution:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

```
struct tuple_wrapper<int, float> {
public:
    using storage_t = std::tuple<int, float>;

    tuple_wrapper(int p0, float p1) : storage_{p0, p1} {}

private:
    storage_t storage_;
};
```

Parameter packs. Expansion contexts

- Instantiation:

```
tuple_wrapper<int, float> w{1, 2.0};
```

int and float are in the pack and expanded according to context

- Substitution:

```
template <typename... Ts>
struct tuple_wrapper {
public:
    using storage_t = std::tuple<Ts...>;

    tuple_wrapper(Ts... args) : storage{args...} {}

private:
    storage_t storage;
};
```

```
struct tuple_wrapper<int, float> {
public:
    using storage_t = std::tuple<int, float>;

    tuple_wrapper(int p0, float p1) : storage_{p0, p1} {}

private:
    storage_t storage_;
};
```

Parameter packs. Expansion contexts

- Expansion pattern will be applied for each item in the pack:

```
template <typename... Ts>
constexpr auto make_tuple_wrapper(Ts && ...args) -> tuple_wrapper<Ts...> {
    return tuple_wrapper<Ts...> { std::forward<Ts>(args)... };
}
```

Parameter packs. Expansion contexts

- Expansion pattern will be applied for each item in the pack:

```
template <typename... Ts>
constexpr auto make_tuple_wrapper(Ts && ...args) -> tuple_wrapper<Ts...> {
    return tuple_wrapper<Ts...> { std::forward<Ts>(args)... };
}
```

Expansion pattern: `std::forward` will be applied to every item in the pack. Packs `Ts` and `args` are expanded simultaneously

Parameter packs. Expansion contexts

- `sizeof...` operator:

```
template <typename T, typename... Ts>
constexpr auto make_array(Ts && ... args) -> std::array<T, sizeof...(Ts)> {
    return { T{std::forward<Ts>(args)}... };
}

constexpr auto array = make_array<int>(123, 456, 'A');
```

Parameter packs. Expansion contexts

- `sizeof...` operator:

```
template <typename T, typename... Ts>
constexpr auto make_array(Ts && ... args) -> std::array<T, sizeof...(Ts)> {
    return { T{std::forward<Ts>(args)}... };
}

constexpr auto array = make_array<int>(123, 456, 'A');
```

Possible implementation of a helper to create `std::array` from values.

Parameter packs. Expansion contexts

- `sizeof...` operator:

```
template <typename T, typename... Ts>
constexpr auto make_array(Ts && ... args) -> std::array<T, sizeof...(Ts)> {
    return { T{std::forward<Ts>(args)}... };
}

constexpr auto array = make_array<int>(123, 456, 'A');
```

Size of a pack defines size of a result array

Parameter packs. Expansion contexts

- `sizeof...` operator:

```
template <typename T, typename... Ts>
constexpr auto make_array(Ts && ... args) -> std::array<T, sizeof...(Ts)> {
    return { T{std::forward<Ts>(args)}... };
}

constexpr auto array = make_array<int>(123, 456, 'A');
```

Expansion pattern initializes objects of type T from each provided argument

Parameter packs. Expansion contexts

- If function parameter pack items can be converted to some common type, they can be used within an initializer list:

```
template <typename T, typename ... Ts>
constexpr auto sum(T init_val, Ts... args) {
    for (const auto i : { T{args}... }) {
        init_val += i;
    }

    return init_val;
}

static_assert(sum(1, 2, 3) == 6);
```

Parameter packs. Expansion contexts

- If function parameter pack items can be converted to some common type, they can be used within an initializer list:

```
template <typename T, typename ... Ts>
constexpr auto sum(T init_val, Ts... args) {
    for (const auto i : { T{args}... }) {
        init_val += i;
    }

    return init_val;
}

static_assert(sum(1, 2, 3) == 6);
```

Type of initial value is used as expected common type

Parameter packs. Expansion contexts

- If function parameter pack items can be converted to some common type, they can be used within an initializer list:

```
template <typename T, typename ... Ts>
constexpr auto sum(T init_val, Ts... args) {
    for (const auto i : { T{args}... }) {
        init_val += i;
    }

    return init_val;
}

static_assert(sum(1, 2, 3) == 6);
```

Can be `constexpr` since C++14

Parameter packs. Expansion contexts

- Expansion within initializer can utilize comma operator to perform complex operations:

```
template<typename... Ts>
auto print_args(Ts... args) {
    (void)std::initializer_list<int>{ (std::cout << args << "\n", 0)... };
}
```

Parameter packs. Expansion contexts

- Expansion within initializer can utilize comma operator to perform complex operations:

```
template<typename... Ts>
auto print_args(Ts... args) {
    (void)std::initializer_list<int>{ (std::cout << args << "\n", 0)... };
}
```

Operation to perform on each item of the pack. Can be function/lambda invocation

Parameter packs. Expansion contexts

- Expansion within initializer can utilize comma operator to perform complex operations:

```
template<typename... Ts>
auto print_args(Ts... args) {
    (void)std::initializer_list<int>{ (std::cout << args << "\n", 0)... };
}
```

Comma operator returns value of a second operand

Parameter packs. Expansion contexts

- Expansion within initializer can utilize comma operator to perform complex operations:

```
template<typename... Ts>
auto print_args(Ts... args) {
    (void)std::initializer_list<int>{ (std::cout << args << "\n", 0)... };
}
```

Array can also be used

Parameter packs. Expansion contexts

- Lambda capture list:

```
template <typename Scheduler, typename... Ts>
void print_args_async(Scheduler scheduler, Ts... args) {
    scheduler.schedule(
        [args...] {
            do_something(args...);
        }
    );
}
```

Parameter packs. Expansion contexts

- Lambda capture list:

```
template <typename Scheduler, typename... Ts>
void do_something_async(Scheduler scheduler, Ts... args) {
    scheduler.schedule(
        [args...] {
            do_something(args...);
        }
    );
}
```

Arguments are captured and used inside lambda

Parameter packs. Expansion contexts

- Lambda capture list:

```
template <typename Scheduler, typename... Ts>
void do_something_async(Scheduler scheduler, const Ts&... args) {
    scheduler.schedule(
        [&args...] {
            do_something(args...);
        }
    );
}
```

Arguments, provided by reference, are captured by reference. Lifetime should be taken into consideration

Parameter packs. Expansion contexts

- Lambda capture list:

```
template <typename Scheduler, typename... Ts>
void do_something_async(Scheduler scheduler, Ts&&... args) {
    scheduler.schedule(
        [...args = std::move(args)] {
            do_something(std::move(args)...);
        }
    );
}
```

Moving the pack into lambda

Parameter packs. Expansion contexts

- Lambda capture list:

```
template <typename Scheduler, typename... Ts>
void do_something_async(Scheduler scheduler, Ts&&... args) {
    scheduler.schedule(
        [...args = std::move(args)] {
            do_something(std::move(args)...);
         }
    );
}
```

`args` is a capture name and can be different

Parameter packs. Expansion contexts

- `alignas` specifier:

```
template <typename... Ts>
class some_variant {
//...
private:
    alignas(Ts...) std::byte buffer[std::max({sizeof(Ts)...})];
};
```

Parameter packs. Expansion contexts

- `alignas` specifier:

```
template <typename... Ts>
class some_variant {
//...
private:
    alignas(Ts...) std::byte buffer[std::max({sizeof(Ts)...})];
};
```

Alignment requirements for each type in the pack will be applied for the declaration

Parameter packs. Expansion contexts

- `alignas` specifier:

```
template <typename... Ts>
class some_variant {
//...
private:
    alignas(Ts...) std::byte buffer[std::max({sizeof(Ts)...})];
};
```

- Instantiation:

```
some_variant<int, float> w;
```

- Substitution:

```
//...
private:
    alignas(int) alignas(float) std::byte buffer[std::max({sizeof(int), sizeof(float)})];
};
```

Parameter packs. Expansion contexts

- `alignas` specifier:

```
template <typename... Ts>
class some_variant {
//...
private:
    alignas(Ts...) std::byte buffer[std::max({sizeof(Ts)...})];
};
```

- Instantiation:

```
some_variant<int, float> w;
```

- Substitution:

Expanded into multiple specifiers

```
//...
private:
    alignas(int) alignas(float) std::byte buffer[std::max({sizeof(int), sizeof(float)})];
};
```

Parameter packs. Expansion contexts

- `alignas` specifier:

```
template <typename... Ts>
class some_variant {
//...
private:
    alignas(Ts...) std::byte buffer[std::max({sizeof(Ts)...})];
};
```

- Instantiation:

```
some_variant<int, float> w;
```

Expansion in the initializer list
`sizeof()` is a pattern

- Substitution:

```
//...
private:
    alignas(int) alignas(float) std::byte buffer[std::max({sizeof(int), sizeof(float)})];
};
```

Parameter packs. Expansion

- Class bases and using statement (example from *cppreference*):

```
template<class... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

- Usage:

```
std::variant<int, long, double, std::string> v{/*...*/};

std::visit(overloaded{
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; }
        }, v);
```

Parameter packs. Expansion

- Class bases and using statement (example from *cppreference*):

```
template<class... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

- Usage:

```
std::variant<int, long, double, std::string> v{/*...*/};

std::visit(overloaded{
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; }
        }, v);
```

Aggregates several lambdas that can be used in `std::visit` call

Parameter packs. Expansion

- Class bases and using statement (example from *cppreference*):

```
template<class... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

Template argument deduction guide specifies that types of lambdas (every lambda has unique type) used for initialization will be template arguments

- Usage:

```
std::variant<int, long, double, std::string> v{/*...*/};

std::visit(overloaded{
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; }
        }, v);
```

Parameter packs. Expansion

- Class bases and using statement (example from *cppreference*):

```
template<class... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

Inherits lambda types and exposes their `operator()`

Pack can also be expanded in member initializer list of constructor, to initialize bases, if needed

- Usage:

```
std::variant<int, long, double, std::string> v{/*...*/};

std::visit(overloaded{
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; }
        }, v);
```

C++17. Fold expressions

- Common pattern to handle parameter packs is recursion:

```
template <typename H>
constexpr auto sum(H head) -> H {
    return head;
}

template <typename H, typename ... T>
constexpr auto sum(H head, T... tail) -> H {
    return head + sum(tail...);
}
```

- Or expansion in initializers (possibly with subexpressions):

```
template <typename T, typename ... Ts>
constexpr auto sum(T init_val, Ts... args) {
    for (const auto i : { T{args}...} ) {
        init_val += i;
    }
    return init_val;
}
```

C++17. Fold expressions

- C++17 fold expressions are designed to simplify pack handling:

```
template<typename T, typename... Ts>
constexpr auto sum(T init_val, Ts... args) {
    return (init_val + ... + args);
}
```

C++17. Fold expressions

- C++17 fold expressions are designed to simplify pack handling:

```
template<typename T, typename... Ts>
constexpr auto sum(T init_val, Ts... args) {
    return (init_val + ... + args);
}
```

Binary left fold (. . . is on the left)

C++17. Fold expressions

- Instantiation:

```
static_assert(sum(1, char{2}, long{3}) == 6);
```

- Substitution:

```
template<typename T, typename... Ts>
constexpr auto sum(T init_val, Ts... args) {
    return (init_val + ... + args);
}
```

```
constexpr auto sum<int, char, float>(
    int init_val, char arg1, long arg2) {

    return ((init_val + arg1) + arg2);
}
```

C++17. Fold expressions

- Instantiation:

```
static_assert(sum(1, char{2}, long{3}) == 6);
```

- Substitution:

```
template<typename T, typename... Ts>
constexpr auto sum(T init_val, Ts... args) {
    return (init_val + ... + args);
}
```

```
constexpr auto sum<int, char, float>(
    int init_val, char arg1, long arg2) {

    return ((init_val + arg1) + arg2);
}
```

Binary left fold applied

C++17. Fold expressions

- Allow to apply all binary operations to items of parameter packs
- Fold expression types

Unary right fold	(pack OP ...)	(element1 OP (element2 OP (... OP elementN)))
Unary left fold	(... OP pack)	(((element1 OP element2) op ...) OP elementN)
Binary right fold	(pack op ... op value)	(element1 op (element2 op (... op (elementN op value))))`
Binary left fold	(value op ... op pack)	(((value op element1) op element2) OP ... OP elementN)

C++17. Fold expressions

- Unary fold expression can be used where applicable:

```
template<typename... Ts>
constexpr auto sum(Ts... args) {
    return (args + ...);
}
```

C++17. Fold expressions

- Unary fold expression can be used where applicable:

```
template<typename... Ts>
constexpr auto sum(Ts... args) {
    return (args + ...);
}
```

Not clear from interface that it doesn't work for empty pack

C++17. Fold expressions

- Unary fold expression can be used where applicable:

```
template<typename... Ts>
constexpr auto sum(Ts... args) {
    return (args + ...);
}
```

Fold with '+' doesn't support empty packs

C++17. Fold expressions

- The following operators are allowed for empty packs:

  `&&` evaluates to `true`

  `||` evaluates to `false` and

  `,` evaluates to `void`

C++17. Fold expressions

- Fold with logical operations:

```cpp
template <typename... Ts>
constexpr auto all_are_true(const Ts& ...args) {
    return (args && ...);
}

static_assert(all_are_true() == true);

template <typename... Ts>
constexpr auto at_least_one_is_true(const Ts& ...args) {
    return (args || ...);
}

static_assert(at_least_one_is_true() == false);
```

C++17. Fold expressions

- Fold with logical operations:

```
template <typename... Ts>
constexpr auto all_are_true(const Ts& ...args) {
    return (args && ...);
}

static_assert(all_are_true() == true);

template <typename... Ts>
constexpr auto at_least_one_is_true(const Ts& ...args) {
    return (args || ...);
}

static_assert(at_least_one_is_true() == false);
```

Result for empty pack

C++17. Fold expressions

- Operation for each item in the pack:

```
template<typename... Ts>
auto delete_all(const Ts&...args) {
    for (const auto &p : {std::filesystem::path{args}...})
        std::filesystem::remove(p);
}
```

- Usage:

```
delete_all("some/file", config.another_file);
```

C++17. Fold expressions

- Operation for each item in the pack:

```
template<typename... Ts>
auto delete_all(const Ts&...args) {
    for (const auto &p : {std::filesystem::path{args}...})
        std::filesystem::remove(p);
}
```

- Usage:

```
delete_all("some/file", config.another_file);
```

no temporary container or initializer list needed for compile time arguments

C++17. Fold expressions

- Operation for each item in the pack:

```
template<typename... Ts>
auto delete_all(const Ts&...args) {
    for (const auto &p : {std::filesystem::path{args}...})
        std::filesystem::remove(p);
}
```

`std::initializer_list` with paths

- Usage:

```
delete_all("some/file", config.another_file);
```

C++17. Fold expressions

- Fold expression with comma:

```
template<typename... Ts>
auto delete_all(const Ts&...args) {
    (std::filesystem::remove(args), ...);
}
```

- Usage:

```
delete_all("some/file", config.another_file);
```

C++17. Fold expressions

- Fold expression with comma:

```
template<typename... Ts>
auto delete_all(const Ts&...args) {
    (std::filesystem::remove(args), ...);
}
```

apply operation for each item pack using fold with comma

- Usage:

```
delete_all("some/file", config.another_file);
```

C++17. Fold expressions

- Perform complex operations:

```
template<typename... Ts>
auto delete_all_report(const Ts&...args) {
    auto delete_and_report = [](auto const &path) {
        if (std::filesystem::remove(path))
            std::cout << path << " was removed" << std::endl;
        else
            std::cout << path << " is not found" << std::endl;
    };

    (delete_and_report(args), ...);
}
```

C++17. Fold expressions

- Perform complex operations:

```
template<typename... Ts>
auto delete_all_report(const Ts&...args) {
    auto delete_and_report = [](auto const &path) {
        if (std::filesystem::remove(path))
            std::cout << path << " was removed" << std::endl;
        else
            std::cout << path << " is not found" << std::endl;
    };

    (delete_and_report(args), ...);
}
```

complex operations can be extracted into lambda (possibly immediate) or function

Fold expressions

- Complex logic in fold expressions (C++17):

```
template <typename F, typename... Ts>
constexpr auto find_first_if(F f, Ts... args) {
    std::common_type_t<Ts...> result{};
    (void)((f(args) ? (result = args, true) : false) || ...);
    return result;
}

static_assert(find_first_if([](const auto it) { return it == 3;}, 1, 2, 3) == 3);
```

Fold expressions

- Complex logic in fold expressions (C++17):

```
template <typename F, typename... Ts>
constexpr auto find_first_if(F f, Ts... args) {
    std::common_type_t<Ts...> result{};
    (void)((f(args) ? (result = args, true) : false) || ...);
    return result;
}

static_assert(find_first_if([](const auto it) { return it == 3;}, 1, 2, 3) == 3);
```

`std` facility to derive common type for parameter pack. It should be default constructible

Fold expressions

- Complex logic in fold expressions (C++17):

```
template <typename F, typename... Ts>
constexpr auto find_first_if(F f, Ts... args) {
    std::common_type_t<Ts...> result{};
    (void)((f(args) ? (result = args, true) : false) || ...);
    return result;
}

static_assert(find_first_if([](const auto it) { return :
```

Ternary operator with comma

Fold expressions

- Complex logic in fold expressions (C++20):

```
template <typename F, typename... Ts>
constexpr auto find_first_if(F f, Ts... args) {
    std::optional<std::common_type_t<Ts...>> result;
    (void)((f(args) ? (result = args, true) : false) || ...);
    return result;
}

static_assert(find_first_if([](const auto it) { return it == 3;}, 1, 2, 3).value() == 3);
```

Fold expressions

- Complex logic in fold expressions (C++20):

```
template <typename F, typename... Ts>
constexpr auto find_first_if(F f, Ts... args) {
    std::optional<std::common_type_t<Ts...>> result;
    (void)((f(args) ? (result = args, true) : false) || ...);
    return result;
}

static_assert(find_first_if([](const auto it) { return
```

`std::optional` can be used in `constexpr` since C++20

Storing parameter packs

- Type list (C++11 and later):

```
template <typename... Ts>
struct type_list {};
```

- Usage:

```
using types = type_list<int, float, std::string>;
```

Storing parameter packs. Application

- Check that the type is supported:

```
template <typename ... EventTs>
struct scheduler {
//...
    using supported_events = type_list<EventTs...>;
};


template <typename SchedulerT, typename EventT>
auto schedule(SchedulerT &scheduler, EventT event) {
    static_assert(is_in_list<EventT>(typename SchedulerT::supported_events{}));
    //...
}
```

Storing parameter packs. Application

- Check that the type is supported:

```
template <typename ... EventTs>
struct scheduler {
//...
    using supported_events = type_list<EventTs...>;
};


template <typename SchedulerT, typename EventT>
auto schedule(SchedulerT &scheduler, EventT event) {
    static_assert(is_in_list<EventT>(typename SchedulerT::supported_events{}));
    //...
}
```

Scheduler supports some event types

Storing parameter packs. Application

- Check that the type is supported:

```
template <typename ... EventTs>
struct scheduler {
//...
    using supported_events = type_list<EventTs...>;
};


template <typename SchedulerT, typename EventT>
auto schedule(SchedulerT &scheduler, EventT event) {
    static_assert(is_in_list<EventT>(typename SchedulerT::supported_events{}));
    //...
}
```

Function uses scheduler to schedule an event

Storing parameter packs. Application

- Check that the type is supported:

```
template <typename ... EventTs>
struct scheduler {
//...
    using supported_events = type_list<EventTs...>;
};


template <typename SchedulerT, typename EventT>
auto schedule(SchedulerT &scheduler, EventT event) {
    static_assert(is_in_list<EventT>(typename SchedulerT::supported_events{}));
    //...
}
```

Check that event type is supported by the scheduler. Can also be used for `enable_if` or `requires` expressions

Metaprogramming evolution

- Utility to check that type is in the list:

```
template<typename T, typename... Ts>
constexpr bool is_in_list(type_list<Ts...>) {
    return is_in_pack<T, Ts...>();
}
```

Metaprogramming evolution

- Utility to check that type is in the list:

```
template<typename T, typename... Ts>
constexpr bool is_in_list(type_list<Ts...>) {
    return is_in_pack<T, Ts...>();
}
```

`is_in_list` is a function template parameterized with pack of types

Metaprogramming evolution

- Utility to check that type is in the list:

```
template<typename T, typename... Ts>
constexpr bool is_in_list(type_list<Ts...>) {
    return is_in_pack<T, Ts...>();
}
```

When instantiated `...Ts`, matches types from `type_list<>`

Metaprogramming evolution

- Check that a type is in a pack. Implementation with recursion (since C++11):

```
template<typename T>
constexpr auto is_in_pack() {
    return false;
}

template<typename T, typename First, typename... Rest>
constexpr bool is_in_pack() {
    return std::is_same<T, First>::value || is_type_in_pack<T, Rest...>();
}

static_assert(is_in_pack<int, double, char, int>(), "int is in the pack");
```

Metaprogramming evolution

- Check that a type is in a pack. Implementation with recursion (since C++11):

```
template<typename T>
constexpr auto is_in_pack() {
    return false;
}

template<typename T, typename First, typename... Rest>
constexpr bool is_in_pack() {
    return std::is_same<T, First>::value || is_type_in_pack<T, Rest...>();
}

static_assert(is_in_pack<int, double, char, int>(),          is in the pack");
```

Function overload handling one or more types. Matches first type and forwards rest to the recursive call if no match

Metaprogramming evolution

- Loops in constexpr (since C++14):

```
template<typename T, typename... Ts>
constexpr auto is_in_pack() {
    for (auto const v : {std::is_same<T, Ts>::value...}) {
        if (v)
            return true;
    }

    return false;
}

static_assert(is_in_pack<int, double, char, int>(), "int is in the pack");
```

Metaprogramming evolution

- Loops in constexpr (since C++14):

```
template<typename T, typename... Ts>
constexpr auto is_in_pack() {
    for (auto const v : {std::is_same<T, Ts>::value...}) {
        if (v)
            return true;
    }

    return false;
}

static_assert(is_in_pack<int, double, char, int>(),
```

No recursion. Transition from types to values

Storing parameter packs

- Fold expressions (since C++17):

```
template<typename T, typename... Ts>
constexpr auto is_in_pack() {
    return (std::is_same_v<T, Ts> || ...);
}
```

- Variable template:

```
template<typename T, typename... Ts>
constexpr bool is_in_pack_v = (std::is_same_v<T, Ts> || ...);
```

Storing parameter packs

- Fold expressions (since C++17):

```
template<typename T, typename... Ts>
constexpr auto is_in_pack() {
    return (std::is_same_v<T, Ts> || ...);
}
```

One liner using ||

- Variable template:

```
template<typename T, typename... Ts>
constexpr bool is_in_pack_v = (std::is_same_v<T, Ts> || ...);
```

Storing parameter packs

- Fold expressions (since C++17):

```
template<typename T, typename... Ts>
constexpr auto is_in_pack() {
    return (std::is_same_v<T, Ts> || ...);
}
```

- Variable template:

```
template<typename T, typename... Ts>
constexpr bool is_in_pack_v = (std::is_same_v<T, Ts> || ...);
```

Can be done as a variable template

`std::tuple` and template parameter packs

- `std::tuple` can store value for each type in the parameter pack:

```
template <typename... Ts>
using storage_for_all_types = std::tuple<Ts...>;
```

- `std::apply` can be used to access tuple elements as parameter pack

```
storage_for_all_types values {/*...*/}
std::apply(
    []<typename... Ts>(Ts&... items) {
        (do_something<Ts>(items), ...);
    },
values);
```

- Has facilities suitable for metaprogramming:

```
using second_item_type = std::tuple_element_t<1, storage_for_all_types>;

constexpr size_t number_of_items = std::tuple_size<storage_for_all_types>;
```

`std::tuple` and template parameter packs

- `std::tuple` can store value for each type in the parameter pack:

```
template <typename... Ts>
using storage_for_all_types = std::tuple<Ts...>;
```

Pack expanded as template arguments

- `std::apply` can be used to access tuple elements as parameter pack

```
storage_for_all_types values {/*...*/}
std::apply(
    []<typename... Ts>(Ts&... items) {
        (do_something<Ts>(items), ...);
    },
values);
```

- Has facilities suitable for metaprogramming:

```
using second_item_type = std::tuple_element_t<1, storage_for_all_types>;

constexpr size_t number_of_items = std::tuple_size<storage_for_all_types>;
```

`std::tuple` and template parameter packs

- `std::tuple` can store value for each type in the parameter pack:

```
template <typename... Ts>
using storage_for_all_types = std::tuple<Ts...>;
```

- `std::apply` can be used to access tuple elements as parameter pack

```
storage_for_all_types values {/*...*/}
std::apply(
    []<typename... Ts>(Ts&... items) {
        (do_something<Ts>(items), ...);
    },
values);
```

Pack refers to tuple items

- Has facilities suitable for metaprogramming:

```
using second_item_type = std::tuple_element_t<1, storage_for_all_types>;

constexpr size_t number_of_items = std::tuple_size<storage_for_all_types>;
```

`std::tuple` and template parameter packs

- `std::tuple` can store value for each type in the parameter pack:

```
template <typename... Ts>
using storage_for_all_types = std::tuple<Ts...>;
```

- `std::apply` can be used to access tuple elements as parameter pack

```
storage_for_all_types values {/*...*/}
std::apply(
    []<typename... Ts>(Ts&... items) {
        (do_something<Ts>(items), ...);
    },
values);
```

Explicit template parameter list (since C++20) can be used to retrieve types of elements. `decltype` can be used instead

- Has facilities suitable for metaprogramming:

```
using second_item_type = std::tuple_element_t<1, storage_for_all_types>;

constexpr size_t number_of_items = std::tuple_size<storage_for_all_types>;
```

`std::tuple` and template parameter packs

- `std::tuple` can store value for each type in the parameter pack:

```
template <typename... Ts>
using storage_for_all_types = std::tuple<Ts...>;
```

- `std::apply` can be used to access tuple elements as parameter pack

```
storage_for_all_types values {/*...*/}
std::apply(
    []<typename... Ts>(Ts&... items) {
        (do_something<Ts>(items), ...);
    },
values);
```

- Has facilities suitable for metaprogramming:

Facilities to inspect lambda properties

```
using second_item_type = std::tuple_element_t<1, storage_for_all_types>;

constexpr size_t number_of_items = std::tuple_size<storage_for_all_types>;
```

Variadic function templates in C++20 abbreviated form

- Function template:

```
template<typename ... Ts>
auto print(Ts... args) -> void { /*...*/ }
```

- Abbreviated form:

```
auto print(auto... args) -> void { /*...*/ }
```

Variadic function templates in C++20 abbreviated form

- Function template:

```
template<typename ... Ts>
auto print(Ts... args) -> void { /*...*/ }
```

- Abbreviated form:

```
auto print(auto... args) -> void { /*...*/ }
```

Uses keyword `auto`

Variadic templates and C++20 concepts

- Variadic class template constrained with concept:

```
template<SomeConcept... Ts>
class Foo {};
```

- Constrained function template:

```
template<SomeConcept... Ts>
auto print(Ts... args) -> void { /*...*/ }
```

- Abbreviated form:

```
auto print(SomeConcept auto... args) -> void { /*...*/ }
```

Variadic templates and C++20 concepts

- Variadic class template constrained with concept:

```
template<SomeConcept... Ts>
class Foo {};
```

- Constrained function template:

```
template<SomeConcept... Ts>
auto print(Ts... args) -> void { /*...*/ }
```

- Abbreviated form:

```
auto print(SomeConcept auto... args) -> void { /*...*/ }
```

Name of used concept

Variadic templates and C++20 concepts

- The following code is not valid C++:

```
constexpr auto sum(int... args) {
    return (0 + ... + args);
}
```

Variadic templates and C++20 concepts

- The following code is not valid C++:

```
constexpr auto sum(int... args) {
    return (0 + ... + args);
}
```

Variadic arguments of distinct type are not supported

Variadic templates and C++20 concepts

- Template parameters are constrained to be all of the type `int`:

```
template <std::same_as<int>... Ts>
constexpr auto sum(Ts... args) {
    return (0 + ... + args);
}
```

- Similar function in the abbreviated form:

```
constexpr auto sum(std::same_as<int> auto... args) {
    return (0 + ... + args);
}
```

Variadic templates and C++20 concepts

- Template parameters are constrained to be all of the type `int`:

```
template <std::same_as<int>... Ts>
constexpr auto sum(Ts... args) {
    return (0 + ... + args);
}
```

Type can be restricted using concepts

- Similar function in the abbreviated form:

```
constexpr auto sum(std::same_as<int> auto... args) {
    return (0 + ... + args);
}
```

Variadic templates and C++20 concepts

- Template parameters are constrained to be all of the type `int`:

```
template <std::same_as<int>... Ts>
constexpr auto sum(Ts... args) {
    return (0 + ... + args);
}
```

- Similar function in the abbreviated form:

```
constexpr auto sum(std::same_as<int> auto... args) {
    return (0 + ... + args);
}
```

In the shorter form

C++26 and variadic templates

- Direct indexing into parameter pack:

```
template <typename... Ts>
void some_function(Ts... args)
{
    using first_type = Ts[0];
    using last_type = Ts[sizeof...(args) - 1];

    const auto first_value = args...[0];
    const auto last_value = args[sizeof...(args) - 1];
}
```

C++26 and variadic templates

- Direct indexing into parameter pack:

```
template <typename... Ts>
void some_function(Ts... args)
{
    using first_type = Ts[0];
    using last_type = Ts[sizeof...(args) - 1];

    const auto first_value = args...[0];
    const auto last_value = args[sizeof...(args) - 1];
}
```

Indexing into template parameter pack (pack of types)

C++26 and variadic templates

- Direct indexing into parameter pack:

```
template <typename... Ts>
void some_function(Ts... args)
{
    using first_type = Ts[0];
    using last_type = Ts[sizeof...(args) - 1];

    const auto first_value = args...[0];
    const auto last_value = args[sizeof...(args) - 1];
}
```

Indexing into function parameter pack (pack of values)

C++26 and variadic templates

- Variadic structured binding:

```
auto print_field_values(const auto &s) {
    const auto &[...items] = s;
    (std::print("{} ", items), ...);
}
```

- Usage:

```
struct some_struct {
    int a;
    float b;
    std::string c;
};

some_struct s {1, 2.0, "hello"};

print_field_values(s);
```

C++26 and variadic templates

- Variadic structured binding:

```
auto print_field_values(const auto &s) {
    const auto &[...items] = s;
    (std::print("{} ", items), ...);
}
```

Variadic structured binding. Pack will correspond to all binded elements

- Usage:

```
struct some_struct {
    int a;
    float b;
    std::string c;
};

some_struct s {1, 2.0, "hello"};

print_field_values(s);
```

C++26 and variadic templates

- Variadic structured binding:

```
auto print_field_values(const auto &s) {
    const auto &[...items] = s;
    (std::print("{} ", items), ...);
}
```

- Usage:

```
struct some_struct {
    int a;
    float b;
    std::string c;
};

some_struct s {1, 2.0, "hello"};

print_field_values(s);
```

Struct instance passed into function

Thank you!

Questions?