Introduction

If some publisher would offer me to write the book that should convince people to start using C instead of C++, it would be as I am asked to talk with dinosaurs – trying to convince them that they are facing with distinction. And I am not Dr. Dolittle – and for sure, I am not on the mission to convince anyone to anything – even if they are walking straight to the cliff. Seriously, it is matter of someone's preferences.

I do love C++ because it is a powerful language.

And the power of the language is measurable in a way how it deals with **complexity** – which kind of tools it provides to abstract the complexity that we try to model, and how efficient these tools are in terms of the expressiveness (the way how they communicate the intention behind) and yes – in terms of the performance as well (and probably fulfilling the other quality goals as well– like portability): the language features that are delivered with std library itself: as our primary toolbox.

One of the shiny features that are heavily used to tame the complexity is template (meta)programming – as the way to write generic, reusable code for solving the same family of the problems, being at the same time highly customizable (like Policy design pattern) – allowing us to use the type-specific solutions with the type-agnostic algorithms.

Variadic parameters pack

Today, I would like to give some brief overview on fold expressions, as the way to apply the same logic on the variadic parameters pack.

You all probably know for __VAR_ARGS__: it is the special identifier used with variadic macros to indicate the arbitrary number of arguments.

You all probably know that with C++ 11 we got the variadic parameter pack – as C++ respond that do not bypass the compiler since it is the language feature that

- Retain the information about the properties of the types: one can use **type-traits/concepts** to inspect each type in the pack on certain properties (like type identity: std::is_a_*<T> traits/concepts), as compile time predicates that one can use:
    - To constraint the template parameter substitution set, limiting the types that can participate in template instantiation (by default, template parameters are unconstrained)
    - To be used with the compile time branching *(if constexpr*), as the way to mimic the function overloading inside the single function – relying on the compiler to remove from the generated code all branches except the one that is evaluated to true (in-place SFINAE)
    - For the class template partial/full specialization, as the way to match the most-specialized one for the given type deduction

- Retain the **value category** (as one of the properties), so that it can be used with perfect forwarding (std::forward) – in case that we talk about the variadic arguments – to properly restore the arguments (including the cv qualifiers) at the calling suite.
- It can be even captured into **std::tuple**: as aggregate type of fixed-size fields, that can be constructed with factory function std::make_tuple() – that takes the variadic parameter pack

Let us demonstrate this on a simple example.
Assume that we want to write the wrapper around the **std::call_once()** that returns the result – only on the first invocation

```
template <typename Function>
struct CallOnce final
{
    private:
        std::once_flag flag_{};
        callable_type callable_{};

    using callable_type = std::decay_t<Function>;
    public:
        template <typename... Args>
        requires std::invocable<callable_type, Args...>
        auto operator()(Args&&... args) noexcept
                            (noexcept(std::invoke<callable_type, Args...>))
        {
            using result_t = std::invoke_result_t<callable_type, Args...>;

            if constexpr (not std::is_void_v<result_t>)
            {
                // will be set only once - the client needs to preserve
                // the result
                std::optional<result_t> result = std::nullopt;
                std::call_once(
                        flag_,
                        [this, &result, ...args=std::forward<Args>(args)]
                        () mutable
                        {
                            result = std::invoke(callable_, args...);
                        }
                );

                return result;

            }
            else
            {
                std::call_once(
                        flag_,
                        callable_,
                        std::forward<Args>(args)...);

            }
        }
```

Before C++20, we could not capture the arguments this way – inside the capture list.
We would instead capture them inside the std::tuple

```
if constexpr (not std::is_void_v<result_t>)
{

    std::optional<result_t> result = std::nullopt;

    std::call_once(
            flag_,
            [this, &result, tuple =
             std::make_tuple(std::forward<Args>(args)...)]() mutable
             {
                result = std::apply([this](auto&&...args) mutable
                          {
                              return std::invoke(callable_,
                                     std::forward<Args>(args)...);
                          }, tuple);
            }
    );

    return result;

}
```

We do not need to check the size of the arguments: sizeof...(Args)
before making the call ***std::make_tuple()****: it handles this internally, by creating an empty std::tuple, and* ***std::apply()*** *knows how to deal with this.*

Ok – this was just to show you the connection between the variadic parameter pack – and the std::tuple.

But in this particular case (as you probably noticed) – this is redundant and **fundamentally wrong** (!!) – since std::call_once() already accepts the callable with *universal signature*

```
std::call_once(
      flag_,
      [this, &result](auto&&...args) mutable
      {
          result = std::invoke(callable_,
                      std::forward<decltype(args)>(args)...);
      }
      , std::forward<Args>(args)...
);
```

<Compiler Explorer>: https://godbolt.org/z/vM6xhhjWd

Fold expressions

The fold expressions are the way to apply the same logic on the given parameter pack – actually, to apply the predefined binary operators along with the comma ‘,’ operator: as the way to apply the custom logic on the parameter pack.

If the fold expression evaluates (expands) parameter pack starting from the left side: **left fold expression**, we can illustrate this as

```
bop(init, bop(…args))

bop(a1, a2) = e1;
bop(e1, a3) = e2;
…
bop(en-1, an) = en;
```

where *bop* is one of the supported binary operators.

The real syntax requires that the entire fold expression is contained inside the pair of parentheses. We distinguish:

- (… op pack) as unary left fold expression, and
- (init op … op pack) as binary left expression.

Now we can write the textbook example

```
constexpr auto sum(std::integral auto ...args) noexcept
{
     // return ( ... + args ); // unary left fold
     return (0 + ... + args); // 0 – init, for proper handling empty pack
}
```

To prevent issue with empty pack – we use binary left fold, with initial value 0, or

```
using comm_type = std::common_type_t<Ts…, int>;
return (comm_type{0} + … + args);
```

This works fine, since integrals are addable types.
For some UDTs - we would need additional check, whether the ***operator+()*** *is defined*

We already mentioned about the concepts and constraints on the type substitution set.
With currently defined constraints, we can write something like

```
details::sum(1, 2, true, false, 3)
```

or – we can be more restrictive and require that all arguments are of the same type

```
template <std::integral A1, std::same_as<A1>...Args>
constexpr auto sum2(A1 a1, Args...args) noexcept
{
   return (a1 + ... + args); // a1 as initial value
}
```

This way – attempt to mix arguments of a different integral types will fail.
The issue with empty pack is here also nice handled – with required first argument as initial value.

<Compiler Explorer>: https://godbolt.org/z/a1EYxWbdh

The parameter pack can be also evaluated from the right end, starting with the rightmost argument: **right fold expression**

```
bop( bop(args…), init)

bop(an, an-1) = e1;
bop(e1, an-2) = e2;
…
bop(en-1, a1) = en;
```

or, using the real syntax

- (pack op …) as unary right fold
- (pack op … op init) as binary right fold

Being able to fold parameter pack from both direction is (from algebra) also known as **Associativity**.
Although the mechanism is the same – the result can be different, for the non-associative operations (like arithmetic: subtraction and division).

<Compiler Explorer>: https://godbolt.org/z/GMEqE4YoE

With this basic introduction, let us analyze this – more interesting peace of the code.
Assume that we have a pool of allocators – stored into the std::tuple

```
using allocator_type = std::tuple<FixedSizeAllocator<BlockSizes,
BlockCount, Alignment>...>;
allocator_type allocators_{};
```

The allocators distinguish themselves on the size of the fixed-size blocks, as internal memory storage (stateful allocators).

On each allocation, we want to find – based on the size requirement, the matching allocator, that will be passed to our allocation function (*func*)

```
template <typename R, typename Func>
requires (std::is_pointer_v<R>)
auto try_alloc(std::size_t size, Func&& func)
{
    R r{nullptr};
    bool found = false;
    std::apply([&](auto&... allocs)
    {
        ((found or (1) (allocs.block_size() >= size and
                    allocs.available_blocks() > 0
                    ? (r = std::invoke(std::forward<Func>(func), allocs),
                       found = true, true) (2)
                    : false)),(3) ...);
    }, allocators_);
```

A few things to notice.

1. Using the std::apply() will invoke the callable on the entire variadic arguments pack – that will be “extracted” by the provided std::tuple, which is inverted procedure than the one we used with std::make_tuple(): converting the variadic arguments pack into the single aggregate type.
2. In this case – we need to traverse over parameter pack, until we find the matching one.
   We accomplish this by relying on the **short-circuit** OR logical operation `(1)`: if the first operand in logical OR expression - *found* flag is true: the second one – the entire ternary operator (?:)

```
(allocs.block_size() >= size and allocs.available_blocks() > 0
 ? (r = std::invoke(std::forward<Func>(func), allocs), found = true,
    true)
 : false))
```

will be discarded: not evaluated (!).

We will still unpack the entire allocators pack - but the second part of the OR expression will not be evaluated, as soon as the first operand becomes true

3. The entire logic is part of the comma operator ',' (3) – so it will be applied iteratively, over the entire variadic pack
4. We have also comma separated operations inside the ternary operator: that will be executed sequentially, one after another – within single fold expression (2)

*The comma (**sequenced**) operator, execute the operations in sequence, discarding the intermediate results, and return the result of the last operation*

```
(r = std::invoke(std::forward<Func>(func), allocs),
 found = true, true)
```

This will obviously invoke our allocation function – on the matching allocator

```
r = std::invoke(std::forward<Func>(func), allocs)
```

The second expression will set the flag to true – which will cause the short-circuit OR expression (1)

```
found = true
```

The third expression is return value for the conditional operator

```
true
```

The rest of the code is not relevant – but for the completeness, in case that we did not find the matching allocator, we will try with the last one – since there is precondition that the blocks – and therefore allocators stored into std::tuple, are sorted ascending by the size of their allocation units (fixed-size blocks)

```
if (not found)
{
    auto& alloc = std::get<SLOTS-1>(allocators_);
    if (alloc.available_blocks() >=
        std::remove_cvref_t<decltype(alloc)>::size2blocks(size))
    {
        r = std::invoke(std::forward<Func>(func), alloc);
    }
}
return r;
```

but only if there is enough memory in remaining available blocks.

This is part of yet another story - so stay tuned (!)

For the demonstration purpose, we can rewrite the code – to make it even more obvious, what is going on under the neath

```
template <typename R, typename Func>
requires (std::is_pointer_v<R>)
```

```
auto try_alloc(std::size_t size, Func&& func)
{

    R r{nullptr};
    bool found = false;

    auto find_first = [&](auto& alloc) mutable
    {
        return (alloc.block_size() >= size and alloc.available_blocks() > 0
               ? (r = std::invoke(std::forward<Func>(func), alloc), found =
                  true, true)
               : false);
    };

    auto apply = [&]<std::size_t...Is>(std::index_sequence<Is...>) mutable
    {
        ((found or find_first(std::get<Is>(allocators_))), ...);
    };

    apply(std::make_index_sequence<SLOTS>{});
```

We define our own search criteria – **find_first()** that returns indication whether the matching allocator is find, and therefore the allocation function being invoked. Then we define our own **apply()** lambda template (since C++20) – parameterized with NTTP variadic pack of indices, that relies, as before, on short-circuit logical OR expression: if find_first() returns true, the subsequent calls will be omitted, while still unpacking the variadic pack – reducing it to the single result.