Expression templates

Montag, 5. Juni 2023 08:32

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

The undisputable fact: C++ doesn't support lazy evaluation of expressions by default, but rather eager computes on the fly.
Expression templates are just one way to have lazy evaluation of an expression, on demand - at the point when it's required.
This way, as with functional programming, we have a declarative way to define the expression, postponing the execution – until it's actually invoked.
With expression templates, we actually bind the callable (expression) with set of arguments (as with std::bind) - as inputs into our expression.
For the same set of the arguments, we can apply (std::apply) different kind of expressions (strategies), which makes the enti re concept – as always when generic programming is involved, highly reusable.
So, the main properties of this pattern are:

- laziness - by evaluation
- intuitive to domain - expressive code (domain specific language)
- reusability

@note It has nothing to do with performance: performance is something that needs to be benchmarked, and the imperative: tailo red code usually outperforms the declarative one, which aims other quality goals (like expressiveness, maintainability, and mentioned reusabil ity).

Usage

We can model an universal expression similar to Bowie Owens nice presentation held on *CppCon 2019*

```
template <typename Func, typename...Args>
class Expression
{
    public:

        using expression_t = Func;

        explicit Expression(Func&&func, const Args&...args) noexcept :
                    expr_(std::forward<Func>(func)),
                    args_(std::tie(args)...)
        {}

       // Call operator - where the expression will be evaluated
        auto operator()() const
        {
            return std::apply(expr_, args_);
        }

    private:
        expression_t expr_;
        std::tuple<const Args&...> args_;
};


// CTDA
template<typename Func, typename...Args>
Expression(Func&&,const Args&...) -> Expression<Func, Args...>;
```

One gotcha with std::tuple as a storage is that it can hold only the distinguish (heterogenous) types.
To cope with this limitation, we need to wrap our arguments of expression into *StrongType*.
More on that at: https://github.com/joboccara/NamedType

Now we need - for having intuitive, expressive syntax, to **overload** the (arithmetic) operators to be used in conjunction with our Expression type.

```
template <typename T1,typename T2>
concept add_binary_expression_on_strong_type = requires(const T1& t1, const T2& t2)
{
    {t1.get() + t2.get() } -> std::same_as<decltype(t1.get() + t2.get())>;
};

/*
 * We return resulting expression - without evaluating it!
 * @note It is first evaluated when we invoke call operator on resulted (returned) Expression
 *       using details::operator+;
 *       auto c = a + b;
 *       std::cout << c() << '\n'; // evaluated first here, invoking call operator!
```

```
 */
template <typename LH, typename RH>
requires add_binary_expression_on_strong_type <LH, RH>
auto operator+ (const LH& lhs, const RH& rhs)
{
    return Expression{[](const auto& arg1, const auto& arg2)
                      { return arg1.get() + arg2.get(); },
                      lhs,
                      rhs
                      };
}
```

As stated in comment, operator+ produces another expression that will be lazily evaluated, first at the point when call operator is called.
The LH an RH can be a different types - or the same type of different dimensions (LH - scalar, RH - vector), as long as they are summable.
To impose this, we need to constraint the parameter types (using concepts).
As matter of fact, they can be different expressions - which is the whole idea with this pattern: to combine different expressions in intuitive and hopefully efficient way.

@note To use it properly with arithmetic operations, with compatible types of different dimensions (and their resulting expressions), we would need instead to equip our generic Expression type with subscript **operator[]**: to access each element of the collection/expression, or return unconditionally the scalar otherwise

```
auto operator[](std::size_t index) const
{
    const auto call_at_index = [this, index](const auto&...args)
    {
        return expr_(subscript(index, args)...);
    };

    return std::apply(args_, call_at_index);
}
```

But more on that on the mentioned talk.

Disclaimer: you may see a lot of resemblance with something that you get for free: with **ranges**.
More on that at: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Ranges.pdf

Code

The full above example: https://godbolt.org/z/3qTMoMYzr

We can simplify the implementation, having binary expression that can be essentially used for building any other expression.
Example of code available at: https://godbolt.org/z/Te3PnTfd4

Links

1) https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Expression-template
2) https://gieseanw.wordpress.com/2019/10/20/we-dont-need-no-stinking-expression-templates/
3) https://www.youtube.com/watch?v=4IUCBx5fIv0