Document Number:	P1012R1
Date:	2020-11-01
Project:	Programming Language C++
Reply-to:	Frank Zingsheim `<f dot zingsheim at gmx dot de>`
Audience:	Evolution

Proposal Ternary Right Fold Expression

Revision 1

I. Motivation

A) Use case

The following example shows a simple use case of a ternary right fold expression.

Consider a function `f` with an index based template parameter:

```
#include <cstddef>

class T;

template <std::size_t i>
T f();
```

The task is now to write a function which calls the appropriate template function for a run time index `j` which has a compile time maximal index `(n-1)`. If `j` is out of bounds an exception should be thrown.

With the proposal from this document it would be possible to write this as follows.

```
#include <functional>
#include <stdexcept>

template <std::size_t... is>
T test_impl(std::size_t j, std::index_sequence<is...>)
{
    return ( (j == is) ? f<is>()
                : ... : throw std::range_error("Out of range") );
}

template <std::size_t n>
T test(std::size_t j)
{
    return test_impl(j, std::make_index_sequence<n>());
}

```

If the implementer of the method is sure that the index `j` is below `n` the function `test_impl` can also be written like follows without a trailing `throw` but with `std::unreachable()` instead (see proposal P0627R3 [4]).

```
template <std::size_t... is>
T test_impl(std::size_t j, std::index_sequence<is...>)
{
    return ( (j == is) ? f<is>() : ... : std::unreachable());
}

```

B) Consistent Completion of Fold Expressions

The proposed syntax is a canonical extension of the already existing fold expression for binary operators [1].
The right fold expansion is applicable to any binary operator which return value can be used as a right argument of the same binary operator.

Since for the conditional ternary operator the return value of the operator can be used as a right argument of the conditional ternary operator, the conditional ternary operator can be expanded in a right fold expression, consistently.

II. Proposed Expansion of Ternary Fold Expression

Only right fold expressions are supported for the conditional ternary operator. Left fold expressions are **not** supported for the conditional ternary operator.

Let `C` denote a non-expanded parameter pack which expand to conditions with `sizeof...(C) == N`.
Let `E` denote a non-expanded parameter pack of the same size as `C`.
Let `D` denote an ordinary expression.

The following fold expression

```
( C ? E : ... : D )
```

expands to

```
( C(1) ? E(1) : ( ... ( C(N-1) ? E(N-1) : ( C(N) ? E(N) : D ) ) ) )
```

The limiting case `N = 0` evaluates to `( D )`.

III. Extension of Conditional Operator

In order to combine the conditional operator [2] easily with the `std::unreachable()` from proposal P0627R3 [4] the handing of void types on conditional operators has to be relaxed.

In C++ 17 the following rule holds: for a conditional operator [2]:

> If either the second or the third operand has type void, one of the following shall hold:
>
> — The second or the third operand (but not both) is a (possibly parenthesized) throw-expression (8.17); the result is of the type and value category of the other. The conditional-expression is a bit-field if that operand is a bit-field.

> — Both the second and the third operands have type void; the result is of type void and is a prvalue.
> [Note: This includes the case where both operands are throw-expressions. — end note]

The relaxed rule would not only allow throw-expressions but also `noreturn` functions [3]. The relaxed rule would read as follows:

> If either the second or the third operand has type void, one of the following shall hold:
>
> — The second or the third operand (but not both) is a (possibly parenthesized) throw-expression (8.17) **or noreturn functions (10.6.8)**; the result is of the type and value category of the other. The conditional-expression is a bit-field if that operand is a bit-field.
>
> — Both the second and the third operands have type void; the result is of type void and is a prvalue.
> [Note: This includes the case where both operands are throw-expressions. — end note]

By this extension the following implementation of a `checked_sqrt` function would be valid:

```
[[noreturn]] void argument_must_be_non_negative(
    std::string_view func_name,
    double x)
{
    auto what_stream = std::stringstream{};
    what_stream <<
        "The argument of " << func_name << " must be non-negative.\n"
        "The function was called with the value: " << x;
    throw std::invalid_argument(what_stream.str());
}

double checked_sqrt(double x)
{
    return (x >= 0) ? sqrt(x) : argument_must_be_non_negative("sqrt", x);
}
```

The implementation of `checked_sqrt` could be rewritten without the conditional operator easily. Or the `argument_must_be_non_negative` function could be given the correct return value (which could be hard for a general generic function since the return type has to be provided as template parameter).

However, the usage of proposed relaxed conditional operator reveals its potential in combination with a fold expression of conditional operators and the `std::unreachable` function from proposal P0627R3 [4]:

```
template <std::size_t... is>
T test_impl(std::size_t j, std::index_sequence<is...>)
{
    return ( (j == is) ? f<is>() : ... : std::unreachable());
}
```

By this it can be expressed that all expected values of `j` are covered by the `is...` and the compiler does not have to add an extra branch for non covered `j` values. The implementation would work since the signature of `std::unreachable` reads as `[[noreturn]] void std::unreachable()`.

VI. Further Example Use Case

Suppose, one has a collection of translation classes defined as follows.

```cpp
#include <string>
#include <string_view>

struct german
{
    static constexpr char language[] = "German";
    static std::string translate_to_english(
        std::string_view text);
};

struct french
{
    static constexpr char language[] = "French";
    static std::string translate_to_english(
        std::string_view text);
};

struct spanish
{
    static constexpr char language[] = "Spanish";
    static std::string translate_to_english(
        std::string_view text);
};

```

The supported languages are known at compile time such that one wants to call the translation like follows.

```cpp
std::string translate_to_english(
    std::string_view language,
    std::string_view text)
{
    return translate_to_english_impl<german, french, spanish>(
        language,
        text);
}
```

The task is now to write the function `translate_to_english_impl` which calls the correct translation with a language string given at run time.

A) Solution with Fold and Throw

This task could be solved with ternary fold expression from this proposal like follows.

```cpp
#include <stdexcept>

template<class... translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return ( language == translators::language
             ? translators::translate_to_english(text)
             : ... : throw std::invalid_argument(
                         std::string("Unknown language: ").append(
                             language.begin(),
                             language.end())) );
}
```

B) Solution with Fold and Noreturn Function

If one wants to factor out the handling of assembling the exception into a function one can do this as follows due to the relaxed rules for the conditional operator proposed in III. Extension of Conditional Operator.

```
#include <stdexcept>

[[noreturn]] void unknown_language(
    std::string_view language)
{
    throw std::invalid_argument(
        std::string("Unknown language: ").append(
            language.begin(),
            language.end()));
}

template<class... translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return ( language == translators::language
             ? translators::translate_to_english(text)
             : ... : unknown_language(language) );
}
```

C) Solution with Fold and Explicit Default

If the first language is the default language this could be realized as follows.

```
template<class default_translator, class... translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return ( language == translators::language
             ? translators::translate_to_english(text)
             : ... : default_translator::translate_to_english(text) );
}
```

D) Solution with Fold and Unreachable

If one wants to tell the compiler that the list of languages is complete (maybe because the argument has already been checked before) this could be done as follows with the `std::unreachable` function proposed in P0627R3 [4] and the relaxed rules for the conditional operator proposed in III. Extension of Conditional Operator.

```cpp
#include <utility>

template<class... translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return ( language == translators::language
             ? translators::translate_to_english(text)
             : ... : std::unreachable() );
}
```

V. Comparison to Alternatives already available in C++20

This paragraph discusses how the functionality of the fold expression in A) Solution with Fold and Throw could be reached with functionality already available since C++20. (The examples can be found on Compiler Explorer https://gcc.godbolt.org/z/qzup48, too.)

A) Explicit Calls

Of cause one always has the possibility to explicitly resolve the fold expression.

```cpp
#include <stdexcept>

template<class translator1, class translator2, class translator3>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return language == translator1::language
           ? translator1::translate_to_english(text)
           : language == translator2::language
           ? translator2::translate_to_english(text)
           : language == translator3::language
           ? translator3::translate_to_english(text)
           : throw std::invalid_argument(
                         std::string("Unknown language: ").append(
                             language.begin(),
                             language.end()));
}
```

This implementation would create exactly the same binary as the fold expression. However, the implementation is less generic since it is limited to a fixed number of languages, in this case three, and it contains duplication of code.

B) Recursion

Fold expressions can often be emulated by recursion. This is also true for the fold of the conditional operator. A recursive implementation would look like:

```cpp
#include <stdexcept>

template<class first_translator>
```

```
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return language == first_translator::language
             ? first_translator::translate_to_english(text)
             : throw std::invalid_argument(
                         std::string("Unknown language: ").append(
                             language.begin(),
                             language.end()));
}

template<class first_translator, class second_translator, class...
translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    return language == first_translator::language
             ? first_translator::translate_to_english(text)
             : translate_to_english_impl<second_translator, translators...>(
                 language,
                 text);
}
```

With the recursion the implementation detail is spread over several function, i.e. the recursion start and the recursive functions.

C) Reuse the fold on operator||

Fold expression can often be emulate by making use of another fold expression. The is also true for the fold of the conditional operator. It can be emulated by the fold on operator|| [5].

```
#include <stdexcept>

template<class... translators>
std::string translate_to_english_impl(
    std::string_view language,
    std::string_view text)
{
    auto translation = std::string{};
    (void)( (language == translators::language
                ? (translation = translators::translate_to_english(text),
                   true)
                : false)
            || ... ||
            (throw std::invalid_argument(
                         std::string("Unknown language: ").append(
                             language.begin(),
                             language.end())), true) );
    return translation;
}
```

Besides from the fact that there is a lot of code which distracts from the original intend. This approach has the following drawback compared to the direct usage of the fold on the conditional operator proposed here.

1. The return type has to be default constructible, move or copy assignable, move or copy constructible (which is the case for `std::string` but is not valid for all types).
2. Additional overhead might be created by calling the default constructor and a move assignment. (Note: The move or copy constructor is not called due to NRVO.)

VI. Design Decisions

A) On not Supporting Fold Expressions without Default Expression

A fold expression without default expression would look like `( C ? E : ... )`. However, this notation leads to confusion since it is unclear what to do with the n-th condition `C(N)` in case `sizeof...(C)` and `sizeof...(E)` is equal to `N`.

This case can be expressed with less confusion by `( C ? E : ... : std::unreachable())`.

The only advantage of `( C ? E : ... )` compared to `( C ? E : ... : std::unreachable())` would be that in the first case the compiler would not call `C(N)` whereas in the second case the compiler has to call `C(N)` in case it may have side effects even though the result is not used anymore.

However, this slight difference may not be worth the additional confusion and the fold expression without default expression could be added in any later C++ standard version if needed.

VII. Acknowledgements

Many thanks to Arthur O'Dwyer for his valuable feedback.

VIII. Revision History

- Revision 0:
  - Initial proposal
- Revision 1:
  - Rename 'initial value' to 'default expression'
  - Remove proposal for ternary fold without initial value
  - Proposal to relax void handling on conditional operator
  - Include usage of Unreachable Code proposal P0627R3 [4]
  - Enhancing examples with throw in last argument of ternary expression
  - Added comparison to alternative implementations already available in C++20

IX. References

[1] Programming Languages - C ++, ISO/IEC 14882:2017(E), 8.1.6 Fold expressions [expr.prim.fold] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.fold

[2] Programming Languages - C ++, ISO/IEC 14882:2017(E), 8.16 Conditional operator [expr.cond] https://timsong-cpp.github.io/cppwp/n4659/expr.cond

[3] Programming Languages - C ++, ISO/IEC 14882:2017(E), 10.6.8 Noreturn attribute [dcl.attr.noreturn] https://timsong-cpp.github.io/cppwp/n4659/dcl.attr.noreturn

[4] Function to mark unreachable code https://wg21.link/P0627R3

[5] foonathan::blog(): Nifty Fold Expression Tricks: Get the nth element (where n is a runtime value) https://foonathan.net/2020/05/fold-tricks/

[6] GitHub repository of this document https://github.com/zingsheim/ProposalTernaryFold