

Document number: DXXXX

Date: 2015-08-19

Project: The C++ Programming Language, Core Working Group

Title: Unary Folds and Empty Parameter Packs, version 2

Reply-to: Thibaut Le Jehan `<lejehan.thibaut@gmail.com>`

Table of Contents

| | | |
|-----|---------------------------------|-------------------|
| I | Introduction | 2 |
| II | Motivation and Scope | 2 |
| III | Alternative solutions | 4 |
| IV | Wording | 9 |
| V | Acknowledgements | 9 |

| | |
|---------------------|---------------------------|
| Bibliography | 10 |
|---------------------|---------------------------|

I Introduction

This document revises N4358 [1]. While the proposed resolution is exactly the same, this paper explores other solutions and explains why they wouldn't work in practice. N4358 aimed to remove from the standard some of the operators from the table "Value of folding empty sequences" proposed in N4295, Folding expressions [2]. We still propose to remove `operator+`, `operator*`, `operator&` and `operator|` from the aforementioned table. The overall goal is to reduce an unexpected and silent behavior of *unary folds* while keeping the design space open for later additions.

II Motivation and Scope

The purpose of allowing empty parameter packs in *unary folds* is to allow users not to have to write binary folds for the simplest cases. However, whatever is the true intent of the users, there is only one specific type which will always be returned for a given operator when the parameter pack in the unary fold is empty. Let us consider the following `sum` function:

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

Writing such a function is easy, and it does what it is expected to do most of the time. However, it will always return the integer 0 when `args` is empty. While generally not a problem, if a function has an overload taking a parameter of the expected return type of `sum` and another overload taking an `int` parameter, it may be a problem. Let us demonstrate it with the following piece of code:

```

VectorType vec = { 1, 2, 3, 4, 5 };
// do things with vec...
vec = sum(some_vecs...);

```

It is common for container classes to overload `operator+` for concatenation. That is for example what `std::string` does. However, some container classes such as Eigen's [3] `Array` may also overload `operator=` to fill container with a given scalar value. With such a class, the piece of code above will do what it is expected to do almost every time, but will silently fill `vec` with 0 when `some_vecs` is empty instead of assigning an empty vector to it, which would be the expected behavior.

This unexpected behavior being silent, finding errors linked to it might be rather difficult. On the other hand, if we decide that the program above is ill-formed when `some_vecs` is empty, the potential problem will be obvious when it arises. Note that, even with that change, simple things remain rather simple:

```

VectorType vec = { 1, 2, 3, 4, 5 };
// do things with vec
// ...
vec = (some_vecs + ... + 0); // Old behavior, new rules,
                           // four more key strokes

```

Since the fix is *that* simple, we consider that removing the special behavior of `operator+` with regards to *unary folds* and empty parameter packs may help to catch silent errors while it won't remove any expressive power to fold expressions. We also propose to remove this special behavior from `operator*`, `operator&` and `operator|` to avoid potential surprises.

That said, we feel that it is worth keeping the special behavior of `operator&&`, `operator||` and `operator,` with *unary folds* and empty parameter packs. You can find the rationale about this choice in the discussion about alternative design solutions to the problem.

III Alternative solutions

Before choosing the resolution proposed above, we explored a range of alternative solutions. Some were based on the thought that it would be possible to have the cake and eat it too, and other were even less flexible than the proposed resolution.

A generic solution based on identity elements

First of all, we analyzed the rationale behind the default values provided when an empty parameter pack is given to an *unary fold*. It's easy to make sense out of the default value assigned to `operator+`: the sum of nothing is nothing where 0 means nothing. Otherwise, it seems that the chosen value for an operation represents the identity element [4] for the magma [5] whose set is the most commonly used together with the operation. That's why addition and multiplication return an integer (0 is the identity element for the integer addition and 1 is the identity element for the integer multiplication), the bitwise operations return unsigned integers and logical operations return boolean values. However, the link between an empty fold and an identity is less obvious when the operator is not `operator+`.

Our first thought was to try to generalize the idea of identity elements to user-defined types for a given operation. The problem is that an empty *unary fold* does not know the type of the elements it is supposed to perform operations on, it only knows about the operation it has to perform. A solution would have been to have a fold expression return an `empty_fold` object when given an empty parameter pack that could have been contextually converted to the identity element for a given type. Here is how such an object could be implemented:

```

template<typename BinaryFunction>
struct empty_fold
{
    template<typename T>
    constexpr operator T() const
    {
        return identity_element<T, BinaryFunction>;
    }
};

```

In this piece of code, `identity_element` is a variable template which can be specialized for any magma for which an identity element makes sense. Also, note that the compiler would have to perform a magical match between an operator in the fold expression and the actual type of `BinaryFunction`, adding more complexity to the solution. One solution would be to use `plus<void>`, `multiplies<void>` and the other function objects from the header `<functional>` to represent these operations. Then, specializing `identity_element` would be done as follows:

```

// Identity element for int and addition
template<>
constexpr int identity_element<int, std::plus<>> = 0;

```

It would allow to write code like this:

```

int a = (integers + ...); // 0
double b = (... * reals); // 1.0
std::complex<float> c = (numbers * ...); // 1.0f + 0.0if
std::string d = (... + strings); // ""s

```

That said, it only solves half of the problem. Every magma does not have an identity element. Some only have a right identity or a left identity element. To cover these cases, it would be rather easy to make an empty *unary left fold* return a `left_identity_element` and an empty *unary right fold* return a `right_identity_element` where these new variable templates are declared in such a way that they can be specialized but will fall back to `identity_element` if

not specialized (a general identity element can serve as both a right identity element left identity element):

```
template<typename T, typename BinaryFunction>
constexpr T left_identity_element<T, BinaryFunction>
    = identity_element<T, BinaryFunction>;

template<typename T, typename BinaryFunction>
constexpr T right_identity_element<T, BinaryFunction>
    = identity_element<T, BinaryFunction>;
```

However, while this solution tries to be as generic as possible, it adds more problems than it actually solves: it means that empty folds are not typed but return a type that is only contextually convertible to other types, possibly leading to silent implicit conversions. Moreover, it would require a way to represent functions as objects for the sole purpose of template specialization and would require a mapping between the supported operators and the equivalent function objects (we used the function objects from `<functional>` as an example, but is that even a good idea?). The behavior would still be surprising with `auto`.

In the end, it does not really solve problems, adds many rules, requires implementers and users to care about identity elements and still lets many cases undefined. The whole thing is too complex and not really useful outside of the mathematical realm. In order to test the thing, we still developed a small library [6] to emulate unary folds with this identity elements mechanism. Anyway... what do identity elements have to do with folds?

Deducing the return type when possible

Another solution was to deduce to return type from the empty fold when possible and to make the program ill-formed when it is not possible. For example, the return type of such an empty fold would have been known:

```
// res is an std::string
auto res = (std::string(args) + ...);
```

This solution would also have worked nice with N4072, Fixed Size Parameter Packs [7] since the proposal allows to write empty parameter packs with a known type:

```
template<std::size_t N>
int sum_ints(int...[N] ints)
{
    return (ints + ...);
}

int a = sum_ints(1, 2, 3); // 6
int b = sum_ints(); // 0, identity element of int
// with addition
```

However, even though we don't have data to back it up, we think that the cases where the type of an empty *unary fold* can be deduced does not represent the majority of the cases. And it would still only work for types that have an identity element, requiring users to know what it means and which magmas actually have an identity element. More rules, little benefit; we didn't think this solution was appropriate either.

Removing more operators from the table

The previous sections explain why we didn't choose a generic or clever solution. Another question is: why didn't we choose a more radical solution, namely deleting the whole table instead of letting three of its lines live?

There are few well-known uses for an overloaded `operator,:` the most common one is for assigning a sequence of values (see Boost.Assign [8] and OpenCV's Mat [9] for example). This kind of assignment should now be achieved with initializer lists, making this use case obsolete. Another use was to bypass the fact that

`operator[]` can only take one parameter; people nowadays can also use initializer lists to “pass” several parameters to `operator[]`. If not implicitly defaulted to `void()`, the default value for `operator`, in a fold expression is rather awkward to write, and not pretty either; people will probably expect `(args , ...)` not to do anything but still be valid when `args` is an empty parameter pack, and we think that it’s how it should be. We wouldn’t consider this to be a surprising behavior.

The story for `operator||` and `operator&&` is a little bit different: these operators are known as *boolean operators* and are generally expected to work with `bool` and only `bool`. Moreover, it is generally considered bad practice to overload these operators since lazy evaluation does not happen anymore when these operators are overloaded; overloading `operator bool` instead is the preferred way to do things. Some compilers such as `g++` can even produce a warning (with `-Weffc++`) when these operators are overloaded for this very reason. We believe that having default values for these operators wouldn’t surprise any user and wouldn’t make the code unclear.

Moreover, folds over `operator||` and `operator&&` respectively correspond to *any* and *all* operations. The standard library functions `std::any_of` and `std::all_of` respectively return *false* and *true* when given an empty range, which means that there already is a precedent in the standard, which lessens the risk of bad surprise. These values also happen to be the respective identity elements for `operator||` and `operator&&`, so everything seems to be consistent enough.

For these reasons, we decided that the default values chosen for `operator||`, `operator&&` and `operator`, were reasonable and not surprising and should therefore be kept in the standard instead of being removed altogether with the other operators.

IV Wording

14.5.3 Variadic templates [temp.variadic]

Delete the following lines from Table N (~~deleted lines in blue~~):

Table N. Value of folding empty sequences

| Operator | Value when parameter pack is empty |
|------------------|------------------------------------|
| * | 1 |
| + | int() |
| & | -1 |
| + | int() |
| && | true |
| | false |
| , | void() |

V Acknowledgements

I would like to thank Jens Maurer, Andrew Sutton and Richard Smith for the feedback about the proposal and the helpful advice.

Bibliography

- [1] T. Le Jehan. N4358, unary folds and empty parameter packs. [Online]. Available: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4358.pdf>
- [2] A. Sutton and R. Smith. N4295, folding expressions. [Online]. Available: <https://isocpp.org/files/papers/n4295.html>
- [3] Eigen c++ template library. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page
- [4] Wikipedia. Identity element. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Identity_element&oldid=626639404
- [5] ——. Magma (algebra). [Online]. Available: https://en.wikipedia.org/w/index.php?title=Magma_%28algebra%29&oldid=670419474
- [6] Morwenn. cpp-fold library. [Online]. Available: <https://github.com/Morwenn/cpp-fold>
- [7] B. Maurice. N4072, fixed size parameter packs. [Online]. Available: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4072.html>

- [8] T. Ottosen. Boost assignment library. [Online]. Available: http://www.boost.org/doc/libs/1_57_0/libs/assign/doc/index.html
- [9] OpenCV mat class documentation. [Online]. Available: http://docs.opencv.org/modules/core/doc/basic_structures.html#Mat