Document number:	P0565R0
Date:	2017-02-02
Project:	Programming Language C++
Audience:	Evolution Working Group
Reply to:	Bengt Gustafsson (bengt dot gustafsson at beamways dot com)

Prefix for operator as a pack generator and postfix operator[] for pack indexing

I. Table of contents

II. Introduction

This proposal allows a subexpression to be a **for** loop which generates a pack-like, one element per loop turn. The for head works as usual and the element value is described by the for body expression. With constexpr range the pack-like is a pack.

A postfix pack indexing operator is also proposed, available only outside pack expansions and fold expressions, including bodies of for expressions.

Both features work together to simplify programming with variadic templates. The following example from Appendix A shows how apply can be implemented with a for expression:

```
template<typename F, typename T> decltype(auto) apply(F f, T& tuple)
{
    return f(for (size_t IX : range::ints(tuple_size_v<pars>)) std::get<IX>(tuple) ...);
}
```

An updated version of this document may be found at: P0565

III. Motivation and scope

Ever since the introduction of variadic templates in C++11 a problem has been that there is so little you can do with them. Even simple tasks like getting value n out of a pack requires quite complicated code which if nothing else wastes a lot of compiler (and thus programmer) time.

To remedy this situation many suggestions have been put forth including prefix and postfix indexing operators using [] or {} type parentheses. The problem is however what to index these operators with, as the indices must be compile time constants and still vary over some range. The current solution to this is to use an index_sequence which most often necessitates the creation of a helper function.

Another type of problem relates to the need of being able to convert regular data structures to packs in order to use them in contexts where packs can be used, such as when calling a function. A prime example is the apply function which allows calling a function with the individual elements of a tuple.

To solve these problems today requires expert level knowledge of variadic templates, perfect forwarding and not least patterns to use the bleak tools of the language to achieve the desired effects. Even after significant training writing such code is slow and error prone, the notorious error messages in template code compounding the problem.

This proposal significantly lowers the threshold for programmer skill required to work with variadic template code and makes the process less tedious and error prone. In addition it is foreseen that it could have a significant positive impact on compile times as the number of template instantiations will go down.

The scope of this proposal is to be able to handle these types of situations with straightforward, imperative style programming. As an effect of not adding rules to forbid the use of runtime variable expressions in the for range the scope of fold expressions is also increased to runtime variable size data structures such as vectors.

A very basic implementation of the parsing part has been made in Clang. No attempt at code gneration has been done.

IV. Impact on the standard

For expression

This feature is a core language chage introducing a version of the for loop that is a part of an expression. As for loops are currently not allowed inside expressions there is no old code that can change meaning. However, as a statement can consist of an expression a rule to prefer a for statement over an expression starting with a for expression is instated.

Postfix pack indexing

This feature is a core language change to allow a postfix operator[] with a constexpr index to be applied to a pack outside pack expansions. As pack names are currently not even allowed to be mentioned outside expansions there are no backwards compatibility issues.

V. Design decisions

For expression

This proposal evolved out of the P0535 proposal regarding prefix operator[] to index and slice packs and tuple-likes. It was observed that even though the slicing operation contained the full Python possibilities with negative and out of bounds indices it still lacked abilities to set a step or for instance reverse the sequence. A pack indexing operator of any syntax is by itself of limited use as the generation of the required constexpr indices is non-trivial. This led to the idea of replacing the slicing of the packs themselves with a possibility to limit the range of the actual pack expansion. The next step was to conclude that being able to use the (constexpr) index of that loop in the expression being expanded would replace the need for slicing and indexing of tuple-likes too.

One design decision was to use the for keyword for this feature. This has the advantage of an intuitive interpretation and lineage from Python for expressions acts as a precendent. An important reason for this decision was the lack of available operator tokens to use for this purpose, and, if such a token could be found, the need to explain why it is not overloadable. Another possibility that was examined was to augment the ... of the pack expansion and fold expression with a way to set its range and introduce a loop variable. One problem with this is that fold expressions already use both sides of the ... for other purposes.

Another design decision was to keep the for as a prefix operator. This creates a slight parsing issue but has the advantage of introducing the loop index before use, rather than after as would be the case for a more Python-esque infix syntax.

It was also decided to allow the full possibilities of the for statement head when it comes to allowing both traditional and range based syntax and allowing both pre-existing and newly declared loop variables as well as multiple variables. Note however as in the constexpr case preexisting loop variables is not an option as these are constexpr and thus can not be changed by the for expression!

It was decided to prefer a solution allowing non-constexpr ranges when the for expression is used in conjunction with a fold expression. The main motivation for this was to make fold expressions more widely useful and to avoid limitations that in time will fall in the "for historical reasons" category.

An alternative design that has been suggested includes a co-routine style pack generator function. Such a function would *yeild* values (possibly of different types) for each element of the pack expansion. One major issue with this type of solution is that inside the generator function a loop with a yield statement would exist. If the type yielded differs between each turn of this loop a completely new rule set regarding compilation of a loop would have to be devised for the entire language. Other problems include possible future interference (in syntax or mindset) with proposals for regular co-routines and the fact that this does not work for runtime variable ranges.

Postfix pack indexing

A major design decision was to use the regular postfix indexing syntax. This minimizes the impact on parsing, but to decide what the indexing operator really means the compiler needs knowledge of its context, i.e. if it is in a pack expansion or not.

The reason for this design decision was mainly that programmers would easily understand what is going on and feel at home with the syntax. The reuse of the postfix operator[] however means that the new meaning can not be allowed in a pack expansion as the construct already has meaning there (i.e. to index each pack element). In general this is not a big problem as the pack expansion can be complemented by a for expression to fully control how indexing of packs is to be done.

It was decided not to offer any direct syntax to create a slice of a pack by somehow specifying starting and ending indices. This decision was based on the fact that for expressions and pack indexing handles most cases where such slices would be useful.

It was decided not to extend pack indexing to non-packs as suggested in P0535 as for expressions handle these cases without having to resort to transforming a built in indexing operator (a language feature) to a `std::get<>` call (a library feature).

Interaction with other proposals and possible proposals

- constexpr range functionality indicated here by the ints() function in the examples section would allow range based for formulations to be used when the for expression is used with a pack expansion. It is unclear to the author to what extent the current ranges proposal will be formulated with constexpr in mind.
- constexpr function parameters as a terse version of non-type template parameters plus addition of operator[](constexpr size_t ix) on tuple and array would allow nicer code than `get<IX>(tuple)` to be used when those types of data are involved.
- Lifting the requirement that fold expressions are enclosed in parentheses. It is at present unclear to the author why this is currently required.
- Changing the parameter of a fold expression from cast-expression to for-expression to remove another parenthesis set. This would in general be a good idea to make fold expressions more similar to the pure pack expansions which have a much lower precedence.

- A (library only) possibility to write `for (auto ix : 5)` is available (see sketch in Appendix C below). Standardizing this would increase the terseness of the typical for expression head, and has other uses with for *statements* .
- Compile time type information proposals based on built in traits for tasks such as enumerating struct member names and types would benefit from the simplicity of the for expression to enumerate through the members.
- Proposals to access struct members via some type of indexing would get help from for expressions to get indexes for their operators.
- While this proposal may reduce their utility there does not seem to be any reason to believe that proposals regarding pack variables or pack return types would be negatively impacted by this proposal.

VI. Technical specification

For expression

A for expression is similar to a for statement but embedded in an expression. A for expression generates what is referred to as a pack-like. The difference from a pack is that it can have a runtime variable number of elements. When the loop range is constexpr the pack-like is a pack and can be used as any pack to initiate array elements, generate function parameters etc. All pack-likes can be used together with fold expressions.

A for expression must be combined with a pack expansion or fold expression that drives the loop and controls what to do with the generated pack elements. A for expression that is not part of a pack expansion or fold expression is ill-formed.

The contents of the parenthesis after the for keyword (the for head) follows the same rules as a for statement head. The body of the for expression is limited to a conditional-expression.

When the loop range is constexpr the for expression body is semantically analyzed separately for each loop turn as in a pack expansion. This allows for heterogeneous types that would be produced by for instance a `std::get<IX>(tuple)` inside the for body. However, in contrast with a pack expansion identifiers denoting packs refer to the whole pack, not the current pack element. This means that for packs to be mentioned in the for expression body they have to be indexed.

When the loop range is not constexpr the code generated for the loop body has to be the same for each turn (as the loop index is not constexpr). This means that the operator overload selected by a fold expression has to be the same so code can easily be generated as a runtime loop. For binary folds the initial operator application where one operand is the init value another operator overload may be selected. Note that if the operator overload selected does not return the same type as its accumulator side operand the fold expression is ill-formed (as this would prevent code from being generated as a loop and thus impossible to generate for a runtime variable index range).

In fold expressions involving a for expression and a shortcut operator (&& or ||) the for loop only runs as many turns as required, even if the for expression range is constexpr. With runtime variable range this can be implemented as if there was a break inside the loop, while with constexpr range code based on branch instructions needs to be generated, just as for any use of shortcut operators.

Syntactical changes

Prefix for is a new operator located between assignment and ?: in precedence, so the assignment-expression production would be changed to this:

```
assignment-expression:
    for-expression
    logical-or-expression assignment-operator initializer-clause
    throw-expression
```

The new functionality is represented by:

```
for-expression:
    conditional-expression
    for ( for-init-statement condition ; expression ) conditional-expression
    for ( for-range-declaration : for-range-initializer ) conditional-expression
```

Note: A rule must be present (which is hard to express in the grammar) that if a statement starts with the *for* keyword it is a for statement. The for of a for-expression can not be the first token of a statement. This rule has no practical implications for programmers as the set of valid for expressions is a subset of the set of valid for statements.

Postfix pack indexing

This feature allows indexing a pack to access one of its elements outside of pack expansions.

In pack expansions this operator is not available as the pack name denotes the current pack element and not the pack itself.

This feature applies to value, type and template packs alike.

There are no effects on the syntax of the language as this construct is already covered in postfix-expression.

VII. Acknowledgements

This proposal created after lengthy email discussions with Matthew Woehlke regarding his P0535 proposal, discussions which drifted in the direction of the present proposal. The for based syntax was also initially suggested by Matthew, by reference to the Python infix form.

VIII. References

P0535. Generalized unpacking

N4235. Selecting from parameter packs

Appendix A. Examples

Here are several examples of how the for expression and in some cases postfix pack indexing can be used, sometimes to do things otherwise not possible, sometimes for convenience.

Apply

The current state of the art implementation of apply uses std::index_sequence, thereby reducing the helper functions to one and the template instantiations to two. Nevertheless it is hard to describe this implementation as straigt-forward.

```
template <typename F, typename T, std::size_t... Is>
constexpr decltype(auto) apply_impl(F&& f, T& tuple, std::index_sequence<Is...>)
{
    return f(std::get<Is>(tuple)...);
}

template<typename F, typename T> auto apply(F&& func, T& tuple)
{
    constexpr size_t sz = std::tuple_size_v<std::decay_t<T>>;
    return apply_impl(std::forward<F>(f), t, std::make_index_sequence<sz>());
}
```

With a for expression no helper function is required:

```
template<typename F, typename T> decltype(auto) apply(F f, T& tuple)
{
```

```
    return f(for (size_t IX : ints(tuple_size_v<pars>)) std::get<IX>(tuple) ...);
}
```

(This code relies on a constexpr ints() function from some range library which generates a range of ints from 0 to its parameter - 1).

Examining this code in detail shows that it uses the standard `get<size_t>` to get each element out of the tuple. The resulting pack is then expanded to a function parameter list. This imposes the requirement that the for expression's range must be constexpr. In this example the same restriction is also imposed by the use of the loop index as a template parameter to `get<>` .

Initializing a C array

Currently you can only initialize a C array with an initializer clause which has an explicit expression for each array element. With for expressions you can do such initializations programmatically.

```
double sin_table[360] = { for (size_t i = 0; i != 360; i++) sin(i * pi / 180) ... };
```

The advantage with this is that the compiler can precompute the table and store int as initialized data in the executable.

It has been noted that there are ways to initiate std::array objects by using their copy constructor and the inclination of compilers to optimize it away. The code to get it done is however less succinct, and could look something like this (thanks to Matthew Woehlke):

```
#include <array>

constexpr double gen sin value(size t index)
{
    return sin(index * pi / 180);
}

template <int... Index> constexpr std::array<double, sizeof...(Index)>
gen_sin_table_helper(std::index_sequence<Values...>)
{
  return {gen_sin_value(Index)...};
}

template <size t Size> constexpr std::array<double, Size> gen sin table()
{
  return gen table helper(std::make index sequence<Size>{});
}

constexpr auto sin_table = gen_sin_table<256>();
```

Reversing a tuple

Reversing a tuple today requires considerable template programming skill. Even with the proposed pack and tuple slicing functionality of P0535 and `if constexpr` it gets quite complex and slow to compile, without those features even more daunting. Here is the P0535 example:

```
template <int n, typename... Args>
auto reverse_tuple_helper(Args... args)
{
    constexpr auto r = sizeof...(args) - n; // remaining elements
    if constexpr (r < 2)
        return make_tuple(args...);

    return reverse tuple helper<n + 1>(args[:n], args[-1], args[n:-1]);
}

template <typename T>
auto reverse_tuple(T tuple)
```

```
{
    return reverse_tuple_helper<0>([:]tuple...);
}
```

This code becomes considerably simpler with a for expression:

```
template <typename T>
auto reverse_tuple(T& tuple)
{
    return make_tuple(for (auto IX : ints(tuple_size_v<T>)) get<tuple_size_v<T> - IX - 1>(tuple) ...);
}
```

Note how the tuple is accessed using the standard get method and does not rely on compiler magic to look for std::get when it finds a special indexing or slicing operator.

Calculating the sum of the elements of a vector

With fold expressions we have got a nice functional style of applying an infix operator over a pack. This functionality would however be equally useful for other containers, including those with runtime variable size. The for expression does not in itself imply restrictions to prevent this as exemplified here:

```
std::vector<double> terms;

double sum = ((for (double t : terms) t) + ...);
```

This example makes the difference between the pack-like and the pack obvious. No pack is in sight but fold expressions are available for all pack-likes generated by for expressions. As a pack is a pack-like the interpretation of fold expression over a pack is the same as before (including handling heterogenous types). As the fold expression operand is only a cast-expression we need an extra set of parentheses around the for.

By definition there must always be a parenthesis around the entire fold expression.

Tuple relational operators

This example shows that you can reuse the loop index and make a more complex for loop body in a straight forward and understandable way. Currently a index_sequence and a helper function would have to be used, just as for apply.

```
template<typnename... Ts> bool operator<(const tuple<Ts...>& lhs, const tuple<Ts...>& rhs)
{
    return ((for (size t IX : ints(sizeof...(Ts))) std::get<IX>(lhs) < std::get<IX>(rhs)) ... &&);
}
```

Again the prolific parentheses are due to the specification of fold expressions.

Defining that the for loop should terminate when the short-circuit operator doesn't need more arguments must be handled which is probably harder in terms of standard text writing than of implementation. There does not seem to be any real ambiguity as the pack-like of a for expression is always generated on the fly and used by the fold expression immediately. It is not possible to "save" the pack-like for later use and reuse which would of course make stopping the loop short impossible. Defining the for expression as a pack generator suggests that if the fold operator doesn't need more data the generator will not be required to produce the excess.

Converting a homogenous vector

This operation, useful in computer graphics, converts a fixed size array of numbers to a 1 shorter array by dividing each of the elements by the last element. To accomplish this we write some imperative code:

This proposal would instead use a for expression. The resulting pack-like is then expanded into a braced initializer as above:

```
template <typename T, size t N>
std::array<T, N-1> normalize(std::array<T, N> a)
{
    return { for (size_t IX : ints(N - 1)) a[IX] / a[N - 1] ...);
}
```

The P0535 proposal solves this by slicing a pack into a one element shorter pack and indexing the pack to produce a value (which the pack expansion does not touch).

```
template <typename T, size t N> std::array<T, N-1>
normalize(std::array<T, N> a)
{
  return {[:-1]a / [-1]a...};
}
```

Here the prefix indexing version is somewhat less verbose but requires learning some new concepts to understand the large difference that the : and the negative indices make... unless you are a seasoned Python programmer.

Slicing a pack

Using a part of a a pack, a so called slice, is where P0535 shines, as this is exactly what that proposal offers.

This can be exemplified by an exceprt from P0535 which improves on a solution from P0478 intended to call `callback()` with the up to five first parameters of `signal()` :

```
// Enormously better
void signal(auto... args)
{
    // pass first 5 arguments to callback; ignore the rest
    callback([:5]args...);
}
```

This competes well compared to what this proposal would offer, but relies on the dubious feature of allowing out of bounds slice indices (in case there are fewer than five arguments to signal). This is not consistents with other parts of the language and hides some types of programmer errors. With this proposal you would use pack indexing and a for expression:

```
// More verbose but easier to understand maybe?
void signal(auto... args)
{
    // pass first up to 5 arguments to callback; ignore the rest
    callback(for (size_t i = 0; i < std::min(sizeof...(args), 5); i++) args[i] ...);
}
```

This code begs the question why args here, seen in both the range and body of the for expression is treated as the *pack* and not the *pack element* even though the subsequent ... operator calls for a pack expansion. The answer is that there are two packs at play, args is the incoming pack that the for expression indexes and the pack generated by the for expression that the ... subsequently expands.

Appendix B: Postfix operator[] for pack indexing, why is it not ambiguous

It has been concluded in N4235 that using postfix operator[] to index a pack would cause ambiguity. This statement is not entirely true, but resolving it in a context where index_sequence is used as the driver for operating on packs and tuples makes certain interesting operations impossible to implement.

Here is the example from N4235:

```
template<typename ... Ts, int ... Ns> std::size_t f() {
  sum(sizeof(Ts[Ns])...);
      // size of Ts[Ns] for each pair or size of Ns-th element of Ts?
}
```

Without postfix pack indexing this can be understood as summing up the sizes of the set of arrays formed as Ts[Ns]. However, it is hard to imagine that even with postfix pack indexing there could be an alternate interpretation as the identifiers Ts and Ns do not stand for packs but pack elements inside the pack expansion.

This means that even with the introduction of postfix pack indexing the current interpretation is the only logical one, and the alternate interpretation offered by the code comment above is not valid.

This would pose a problem for implementations based on index sequences, as you may want to implement a function that really does index Ts with Ns. However, with for expressions there is much more syntax available to express what we really want to do in a clear way. Here is for example the above function's other purported interpretation:

```
template<typename ... Ts, int ... Ns>  // Each N must be smaller than sizeof...Ts.
std::size_t f() {
  sum(for (size_t IX : ints(sizeof...(Ns))) sizeof(Ts[Ns[IX]]) ...);
}
```

Note that this works as the for expression is not part of the outer pack expansion, it generates a pack (of size_t values) that the ... then expands to send to sum(). In the body of the for both packs are unexpanded, and Ns acts as an index table into Ts.

The original example can of course also be written using a for-expression, which would be less terse than the original, and has a somewhat painful formulation of the sizeof() argument. But it works and is not ambiguous, again as the for body is not in a pack expansion.

```
template<typename ... Ts, int ... Ns>  // Packs must be like-sized
std::size_t f() {
    sum(for (size_t IX : ints(sizeof...(Ts))) sizeof(Ts[IX][Ns[IX]]) ...);
}
```

P0535 would look like the original for the current interpretation and like this for the alternate interpretation. Yes the only difference is the placement of the brackets. This causes Ts to be a non-pack in the pack expansion which thus runs sizeof...(Ns) times.

```
template<typename ... Ts, int ... Ns>
std::size_t f() {
    sum(sizeof([Ns]Ts) ...);
}
```

Or that's what this author thought. The author of P0535 claims that this does not work as both Ns and Ts are expanded before the prefix [] is applied, which means that you are taking the size of index Ns of each pack-like Ts parameter. Using Ns as a index table into Ts would need to ensure that the Ts is not expanded, which is done by a few tricks:

```
template<typename ... Ts, int ... Ns>
std::size_t f() {
```

```
    using types = tuple<Ts...>;
    sum(sizeof([Ns]declval(types))...);
}
```

After these exercises it was concluded that with for-expression postfix pack indexing is viable, but without for-expression the usefulness is limited as the disambiguiation that favors the original indexing operator (or in the example actually array declarator) makes it unusable for some purposes.

Appendix C: Simplified integer range creation

Note: This is not part of the proposal, but could if desired be formulated in a separate but somewhat supporting proposal.

By adding std::begin and std::end overloads for integer types the code to write a for loop over a range of integer values starting at 0 can be simplified. This is a library only solution. This would maybe not even be a proposal if it weren't for the fact that range based for only looks for begin functions in the *std* namespace. Here is simplified sample code for this functionality, in particular constexpr handling is lacking.

Last minute update: This is not legitimate C++, so would need a core language change. It did compile on Microsoft VS2015 which fooled the author.

```
namespace std {

    template<typename I> class interator {
    public:
        interator() = default;
        interator(I v) : mVal(v) {}

        // Note the const: This prevents trying to manipulate this index to adjust for a changed
container size.
        const I operator*() { return mVal; }

        interator& operator++() { mVal++; return *this; }  // prefix
        interator operator++(int) { return interator(exchange(mVal, mVal + 1)); }  // postfix
        interator& operator--() { mVal--; return *this; }  // prefix
        interator operator--(int) { return interator(exchange(mVal, mVal - 1)); }  // postfix

        bool operator==(const interator& other) { return mVal == other.mVal; }
        bool operator!=(const interator& other) { return mVal != other.mVal; }
        bool operator<(const interator& other) { return mVal < other.mVal; }
        bool operator<=(const interator& other) { return mVal <= other.mVal; }
        bool operator>=(const interator& other) { return mVal >= other.mVal; }
        bool operator>(const interator& other) { return mVal > other.mVal; }

    private:
        size_t mVal = 0;
    };

    // Unclear which type set is required. This has more to do with avoiding compiler warnings than
functionality.
    // For now just do size t, it works with standard containers.
    inline interator<size t> begin(size t ix) { return interator<size t>(); }
    inline interator<size_t> end(size_t ix) { return interator<size_t>(ix); }
}

// Example:
for (auto ix : 100)
    std::cout << ix << std::endl;
```

The class name will definitely need some bike-shedding!