Document number: ?

Date: 2016-03-01

Project: Programming Language C++, Evolution Working Group

Reply-to: Daniel Frey <d.frey@gmx.de>

Improve variadic parameter packs

Introduction

Most uses of template parameter packs require the pack to be broken down into its constituent elements. For some common simple cases (like "pick the Nth argument") this requires complexity (and compiler resources) disproportionate to the task at hand.

Recursive implementations or indirections are often used, but they have several weaknesses:

- It's expensive in terms of compiler resources (e.g., template instantiations use memory that is never freed during the compilation process, slow compile times).
- The instantiation depth limit is reached even for simple cases (e.g., calling `tuple_cat` with a resulting tuple of only 25 elements reached the limit of 256 when using Clang with libstdc++).
- Diagnostics are likely to be overly verbose.

While recursion can often be avoided, it requires tricks and work-arounds which should not be necessary and *some* overhead will still remain.

The above shows that there is a problem with the fundamentals of handling variadic templates. Simple tasks should be simple and complicated tasks should be possible, but the language currently fails to make the simple tasks actually simple.

It is also not following the zero-overhead principle. In some cases even the generated runtime-code suffers from the additional indirections and recursion and is not as efficient as it should be.

This paper identifies the missing primitives to allow efficient use of template parameter packs, significantly reducing compile-times and memory usage, improve code readability and maintainability, improve error messages and finally improve the generated code in common use-cases.

Proposal

We propose to add three extensions to the language: Singular Expansion, Sequence Generation and Pack Aliases. Those primitives, together with the existing language, combine nicely to significantly improve several common patterns in C++ as will be shown.

Singular Expansion / Selection

Summary: Create only a single expansion of a pattern.
Syntax: *`pattern...`*`[I]`

A typical use-case is found in `std::tuple_element`, allowing the following implementation:

```
template<size_t I, class... Types>
class tuple_element<I, tuple<Types...>>
{
  using type = Types...[I];
};
```

Sequence Generation

Summary: Generate a sequence of length N with values 0 to N-1 of a given integer type.
Syntax: *type*`... N`

A typical use-case is found in `std::make_integer_sequence`, allowing the following implementation:

```
template<class T, T N>
using make_integer_sequence = integer_sequence<T, (T... N)...>;
```

Note that `T... N` generates a non-expanded parameter pack, it needs to be expanded for the above example. Generating a non-expanded pack is often the basis needed for further simplifications.

Pack Aliases

Summary: Allow pack aliases to prevent costly work-arounds.
Example 1: `using Is = size_t... N;`
Example 2: `template<typename... Ts> using Ds = decay_t<Ts>;`

A pack alias is a *pattern* that is not expanded, basically everything that could be expanded by *pattern*`...`.

Example

Take today's implementation of the upcoming `std::apply`:

```
template<class F, class Tuple, size_t... I>
decltype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
  return forward<F>(f)(get<I>(forward<Tuple>(t))...);
}

template<class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& t) {
  using Indices = make_index_sequence<tuple_size<decay_t<Tuple>>::value>;
  return apply_impl(forward<F>(f), forward<Tuple>(t), Indices());
}
```

With Sequence Generation, this can be reduced to:

```
template<class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& t) {
  constexpr auto Size = tuple_size<decay_t<Tuple>>::value;
  return forward<F>(f)(get<size_t... Size>(forward<Tuple>(t))...);
}
```

With no need to create a `std::index_sequence` and an additional forwarder `..._impl` just to deduce an index pack from the just-created `std::index_sequence`.

With Pack Aliases, it can be written as:

```
template<class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& t) {
  constexpr auto Size = tuple_size<decay_t<Tuple>>::value;
  using Is = size_t... Size;
  return forward<F>(f)(get<Is>(forward<Tuple>(t))...);
}
```

Which is more readable and in other cases you might want to use `Is` several times.

Other uses

Other features that are often requested are also covered by the above, e.g., N4235's "Pack Subsetting". This section only combines the above proposed expansions, it does not introduce any other changes to the language.

pattern`...[size_t... 5]...;` // expands to the first five elements of *pattern*

pattern`...[2+size_t... 5]...;` // expands to elements three to seven

pattern`...[4-size_t... 5]...;` // expands to the first five elements in reversed order

```
using Is = 2 * (size_t... 3) + 1; // pack of 1, 3, 5
pattern...[Is]...; // expands to the 2nd, 4th and 6th element of pattern
```

Syntax

The syntax for the first two proposed features uses the ellipsis, since it is nowadays closely tied to variadic templates in most people's minds. By using the ellipsis, we keep things easy to spot and consistent.

The proposed Singular Expansion / Selection is unambiguous since the current language does not allow the ellipsis to be followed by an opening square bracket. No currently valid program will be broken.

The second proposed feature, the Sequence Generation should also not break any existing and valid program as far as I can tell.

The third feature, Pack Aliases, again does not break existing programs as using a pattern containing a pack which is *not* expanded is currently always an error.

Implementability

The proposal should be implementable without introducing new entities to the language. The generated entities are all already-known objects that can currently result from either having a patterns manually expanded once, i.e., it is either a type or a (compile-time) value, or in case of the second and third proposed feature, it is a pack as-if it was generated by argument deduction.

All modifications are therefore in localized areas and do not interact with the rest of the language, compiler, etc. in any new way.