Document number: ?

Date: 2016-03-13

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 `N`-th 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 `std::tuple_cat` with a resulting tuple of 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. Even without recursion, some tasks can not be expressed directly and require unintuitive work-arounds.

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. The "sufficiently-smart compiler" argument is sometimes brought up, put in practice compilers do generate inferior code for several cases.

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 two extensions to the language: Pack Selection and Pack Declarations. Those primitives, together with the existing language, combine nicely to significantly improve several common patterns in C++ as will be shown.

Pack Selection

Create only a single expansion of a pattern.

Syntax: *pattern*`...[`*integral-constant*`]`

Pack Declaration

Allow packs to be declared explicitly.

Syntax 1.1: **`typename...`** `Ts;`
Syntax 1.2: **`typename...`** `Ts =` *`template-argument-list`*`;`
Syntax 1.3: **`using typename...`** `Ts =` *`class`* `::` *`pack`*`;`
Syntax 2.1: *`integral-type`*`... Is;`
Syntax 2.2: *`integral-type`*`... Is =` *`template-argument-list`*`;`
Syntax 2.3: **`using`** *`integral-type`*`... Is =` *`class`* `::` *`pack`*`;`
Syntax 2.4: *`integral-type`*`... Is = ...` *`integral-constant`*`;`

Examples

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

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

A typical use-case for pack declarations is found in the upcoming `std::apply`. The current implementation:

```
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 a pack declaration, this can be reduced to:

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

Syntax 2.4 of pack declarations can be used to implement `std::make_integer_sequence`:

```
template<typename T, T...>
struct integer_sequence;
```

```
template<typename T, T N>
struct make_integer_sequence_impl
{
  T... I = ...N;
  using type = integer_sequence<I...>;
};

template<typename T, T N>
using make_integer_sequence =
  typename make_integer_sequence_impl<T, N>::type;
```

Syntax

The syntax for the 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 pack 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 pack declaration should also not break any existing and valid program as far as I can tell. The proposed pack declarations are always an isolated statement which is easy to spot for the user, uses familiar syntax on both the right hand side as well as the left hand side. We do not allow access to pack members or in-situ creation of parameter packs in order to avoid problems with expansion order and meaning, only by as explicit new syntax (1.3 and 2.3) we allow limited and controlled access to packs that are members of classes.

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 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.