Document number: P2163R0
Date: 2020-05-15
Audience: Language Evolution Working Group
Authors: Mike Spertus, Amazon
Alex Damian, Bloomberg
Reply-To: msspertu@amazon.com, alex@damian.to

Native tuples in C++

1. Abstract

We propose C++ types for native tuples *braced-init-list*s. These types not only provide simpler and more natural notations than `std::tuple` but offer new capabilities like perfect forwarding of *braced-init-list*s, using packs outside of templates, and providing default values for variadic parameters.

The authors would like to acknowledge Barry Revzin for helpful discussions and Matt Godbolt for bringing the variadic logging example to our attention.

2. Problems with the current language idioms

We believe the lack of native tuple types and types for *braced-init-list*s is more than a theoretical deficiency. It degrades many common C++ idioms in ways that occur in real code as illustrated by the following examples. Assume we have the following definitions.

```
struct X {
    X(int i, vector<string> const &vi) : i(i), vs(vs) {}
    int i;
    vector<string> vs;
};
int f(int, X);
int a = 1, b = 2;
```

Problem 1: Imperfect Forwarding

Lacking a type, *braced-init-list*s cannot be forwarded.

In the following code, the user cannot pass the same constructor arguments to heap constructed objects as stack-constructed objects even though it seems like the decision of whether to use automatic or dynamic duration should be independent of the choice of what constructor arguments to pass.

```
X x(3, { 4, "foo"});                        // OK
auto ux = make_unique<X>(3, { 4, "foo" }); // Ill-formed
```

Of course, this is not particular to `make_unique`. It occurs in any construct that relies on perfect forwarding

```
template<typename Callable, typename ...Ts>
auto callDuration(Callable f, Ts ...ts)
{
  auto start = chrono::high_resolution_clock::now;
  f(forward<Ts>(ts)...);
  auto end = chrono::high_resolution_clock::now;
  return end - start;
}

int i = f(1, {2, "x"}); // OK
chrono::duration cd = callDuration(f, 1, {2, "x"}); // Ill-formed
```

Problem 2: Impaired Marshalling

Even when forwarding does not need to be perfect, similar problems persist. Consider the following

```
void g(vector<string>);
g({1'000'000, "foo"s});  // OK
```

Suppose we decide this is too slow to run synchronously and decide to run in another thread with the same arguments

```
thread t(g, {1'000'000, "foo"s}); // Ill-formed. Can't marshall argument to new thread
```

An explicit cast doesn't help either because we would like the time consuming work of building the big vector to take place in the new thread:

```
thread t(g, vector(1'000'000, "foo"s)); // Runs but largely in wrong thread!
```

While the programmer can ultimately work around problems like this by creating a shim function or lambda, from the programmer's point of view, this is a totally arbitrary requirement as other argument types have no such restriction.

Problem 3: Impaired `auto` Return Type Deduction

Without a type, *braced-init-lists* initializers cannot be used with return type deduction

```
auto f1() { return 3; }        // Ok
auto f2() { return {1, 2.3};}  // Ill-formed
```

Problem 4: Suppressed Class Template Argument Deduction

The following natural code is ill-formed even though the intent is unambiguous

```
map arguments = { // ill-formed
    {1, 2.3}, {3, 4.5}, {6, 7.8}
};
```

Problem 5: Surprising Gaps in Structured Binding

In our opinion, the following feels like it should work

```
auto &[a, b] = {i, j};
```

While workarounds are possible, they tend to leverage advanced techniques like `forward_as_tuple`, which we expect will lead to frustration for non-advanced programmers.

Problem 6: CWG1643 - Default Arguments for Template Parameter Packs

According to [CWG 1643],

> Although 13.2 [temp.param] paragraph 9 forbids default arguments for template parameter packs, allowing them would make some program patterns easier to write. Should this restriction be removed?

```
template<typename ...Ts = ???> struct X {}; // Unclear what to even write
```

Problem 7: Parameter Pack Function Parameters Problems

Parameter Packs are commonly used as function parameters for function templates, but often exhibit undesirable behavior when not at the end of the parameter list. Consider the following example (private communication from Matt Godbolt) showing why this prevents us from having a variadic logging function with source locations.

```
template<typename... Ts>
void log(string_view msg, Ts... args, source_location sl = source_location::current())
{   cerr << sl << ": " << format(msg, args...); }

log("Message without args");     // OK
log("foo {}, bar {}", foo, bar); // Ill-formed (assuming bar is not a source_location)
```

Problem 8: `std::tuple` cannot quite emulate a built-in type

In spite of valiant efforts, `std::tuple` cannot really act seamlessly like the sort of built-in tuple type provided by numerous other languages, sometimes leading to subtle differences. For example, we wonder how many programmers realize that a single-type tuple has different conversion behavior than the type it wraps

```
struct A { operator double() { return 1.1; } };
struct B { operator tuple<double>() { return {1.1};} };
void fa(int);
void fb(tuple<int>);

fa(A());  // OK
fb(B());  // Ill-formed
```

Likewise, we suspect that, as a result of its complexity (e.g., 18 constructors!), the compile-time cost of working with tuples is a lot higher than a built-in native tuple would be, whether used as a tuple or a typelist.

3. Native tuple overview

3.1 Basic Usage

The basic idea of the proposal is to make braced initializer lists into literals for a type that decays to a native tuple type of the form (bikeshed) `<Type1, Type2,...>`. We will give the full details below, but our goal is for all of the above use cases to work naturally with at most minimal changes as follows.

```
// Solution 1: More Perfect Forwarding
X x(3, { 4, "foo"});                        // OK
auto ux = make_unique<X>(3, { 4, "foo" }); // Now OK

int i = f(1, {2, "x"}); // OK
chrono::duration cd = callDuration(f, 1, {2, "x"}); // Now OK

// Solution 2: Better Marshalling
g({1'000'000, "foo"s});  // OK
thread t(g, {1'000'000, "foo"s}); // Now OK and constructs vector in  new threaad

// Solution 3: Improved auto Return Type Deduction
auto f1() { return 3; }        // Ok
auto f2() { return {1, 2.3};}  // Now OK. Return type is <int, double>

// Solution 4: Improved Class Template Argument Deduction
map arguments = { // Now OK. Deduces map<int, double> as expected
    {1, 2.3}, {3, 4.5}, {6, 7.8}
};
```

```
// Solution 5: Easier Structured Binding
auto &[a, b] = {i, j}; // Now OK

// Solution 6: Supports Default Arguments for Template Parameter Packs
template<typename ...Ts = ...<int, double> > struct X {}; // Clear solution

// Solution 7: Easier to pass middle parameters variadically
template<typename... Ts>
void log(string_view msg, <Ts...> args = {}, source_location sl = source_location::current())
{   cerr << sl << ": " << format(msg, ...args...); }

log("Message without args");     // OK
log("foo {}, bar {}", {foo, bar}); // Now OK (note slight notation change)

// Solution 8: Built-in tuples convert like built-in types
struct A { operator double() { return 1.1; } };
struct B { operator <double>() { return {1.1}; }; // Now <double> instead of tuple<double>
void fa(int);
void fb(<int>);   // Now <int> instead of tuple<int>

fa(A());  // OK
fb(B());  // Now OK
```

3.2 Native Tuple Types

The most basic native tuple type is bikeshedded as an angle-bracket enclosed list of types, like `<int, ostream>`. One good source of native tuples is *braced-init-lists*. For example,

```
<int, char> n1 = {1, 'x'};
```

See Section 4 for details and more examples.

So what is the type of a *braced-init-list* (note that problems 1, 2, 3, 4, and 7 above require that *brace-init-lists* have a type)? To describe with an example, `{a, "foo"s}` is a type that inherits from `<int &, string>` and decays to `<int, string>`. See section 4.2 for a more comprehensive definition and rationale, but we note here that such a definition makes the following examples work.

```
// Same notation as section 2
auto x1 = {a, "foo"s}; // <int, string>
x1.<0> = 10; // a is not changed. See 3.3 for notation
<int &, string> x2 = {a, "foo"s}; // Avoids decaying reference to a
x2.<0> = 10;  // a now equals 10
auto ux = make_unique<X>(3, { 4, "foo" }); // Forwards as *braced-init-list*
```

As possible extensions, since *braced-init-lists* can have designated initializers, we note that it seems natural to support allowing native tuples to have named fields along with default initializers as long as they meet the requirements of non-type template parameters as in the following example:

```
// Future extension - not yet proposed
<int a, char c = y> n2 = {.a = 4 };
cout << n2.a; // Prints 4
```

See the Future Directions section for further info on supporting field names including descriptions of some technical challenges to be overcome.

3.3 Indexing, slicing, and packing

While separable from the rest of the paper, we also propose language support for indexing and slicing. Just like `std::tuple` is restricted compared to native tuples by what is available to library implementers, the same applies to `std::get`. We therefore propose that `t.<x>` behave the same as `std::get<x>(t)`.

```
<int &, string, double> x = {a, "foo"s, 1.5};
x.<0> = 10;                              // a now equals 10
cout << x2.<string>;                     // Prints foo
```

We also allow slicing of native tuples into subtuples with the following notation.

```
auto x2 = x.<1..3>; // x2 has type <string, double>
```

To facilitate moving safely back and forth between packs and tuple, we allow a tuple to be converted into a pack by preceding if with `...`.

```
    <int &, string, double> x = {a, "foo"s, 1.5};
    cout<< format(message, ...x...);
```

See Section 5 for details and rationale, including interaction with P1858R2, which overlaps with this.

3.4 Working With Packs Outside of Templates

While currently packs are used in conjunction with templates, native tuples make them quite useful outside of templates as well. The following example uses a map to persist possible arguments for a function and later call the function with arguments appropriate to the context.

```
void populateDB(DatabaseHandle, Mode);
enum class Stage { Devel, Staging, Production };
enum class DBMode { Debug, Audit, Performance };
```

```
ostream &operator<<(ostream &os, DBMode d) { /*...*/}

// map<Stage, <DatabaseHandle, DBMode>>
map arguments = { {Stage::Devel,      {DevDB, DBMode::Debug} },
                  {Stage::Staging,    {StagingDB, DBMode::Audit}},
                  {Stage::Production, {ProductionDB, DBMode::Performance}} };

int main()
{
    populateDB(...arguments[currentStage()]...);
    cout << format("Populated DB for {}\n", arguments[currentStage()].<DBMode>);
}
```

4. Declaration and initialization

4.1 Declaration

Let `{T}{cv+ref}...` be an expanded parameter pack (aka `ep`).

Let `{Type}{cv+ref}` be a type declaration (aka `td`).

Let `{ep|td}` be a valid tuple type (aka `tt`).

A native tuple declaration shall always be enclosed in `<>`, and contain zero or more comma-delimited `tt`s as follows:

`<[tt1[, tt2[, ...[, ttN]]]]>`

Ex: `<int, const double&, T..., std::string>` is a legal native tuple.

Native tuples are tuple-like types, making them compatible with `std::get` (although see section 5), structure bindings, etc.

4.2 The type of a *braced-init-list*

While it is tempting to say that a braced initializer has native tuple type, we think this approach has problems. For example, *braced-init-list*s may contain designated initializers that cannot be expressed in native tuple types. Even if this limitation were removed by allowing named fields in native tuples as described in the Future Directions section below, more intrinsic problems would remain.

Suppose we have the following definitions:

```
int i{2};
struct X { X(int &i) { i = 5;} };
void f(X) {}
```

We now consider what the type of `{i}` should be. We claim it cannot be `<int>` because `f({i})` is already valid C++20 code that changes the value of `i` to `5`, and if `{i}` had type `<int>`, `f` would at best change the copy of `i` in `{i}`.

Of course, the above could be fixed by having the type of `{i}` be `<int &>` the way `forward_as_tuple` deduces the template arguments for `tuple`, but this also leads to problems.

```
// Assume decltype({i}) is <int &>
<int &> n0 = {i};
// n1 and n2 must have the same type because their initializers do
auto n1 = {i}; // C++20: n1 has type initializer_list<int>
auto n2 = n0;  // Therefore n2 must be initializer_list<int>
```

But this is very different from what we expect from tuples, which neither decay away the references or turn into initializer lists!

```
tuple<int &> t0 = {i};
auto t2 = t0; // C++20: t2 is tuple<int &>
```

Since `{i}` can be neither `<int>` or `<int &>`, the basic idea is that it be an exposition-only class that inherits from `<int &>` but decays to `<int>`. In particular, the *braced_init_list* `{t1, t2, ..., tn}` is a compiler-generated class satisfying the following

- It inherits from `<decltype(t1), decltype(t2), ..., decltype(tn)>`
- For the purpose of template argument deduction, it decays to `<decay_t<decltype(t1)>, ..., decay_t<decltype(tn)>>` unless all of the `decay_t<decltype(ti)>` are the same, in which case it decays to `initializer_list<decay_t<decltype(t1)>>`.

The below mostly repeats the example from 3.2 with "more understanding":

```
// Same notation as section 2
auto x0 = {a, 7}; // Decays to initializer_list<int> for compatibility
auto x1 = {a, "foo"s}; // <int, string>
x1.<0> = 10; // a is not changed. See 3.3 for notation
<int &, string> x2 = {a, "foo"s}; // Avoids decaying reference to a
x2.<0> = 10;  // a now equals 10
auto ux = make_unique<X>(3, { 4, "foo" }); // Forwards as *braced-init-list*
```

Like native tuples, *braced-init-list*s are tuple-like types.

4.3 Initialization

A native tuple can be converted from another native tuple if the members have the proper convertibility

- If all of the conversions are built-in conversions, the conversion is built-in (this enables solution 8 above)
- Otherwise, if all of the conversions are implicit, the conversion is implicit
- Otherwise, the conversion is explicit

```
// Extracted from Solution 8: Built-in tuples convert like built-in types
struct B { operator <double>() { return {1.1}; };
void fb(<int>);
fb(B());  // OK because double => int is built-in
struct C {
    C(int) {}
    explicit C(char const *) {}
};
<int, C> c1 = {1, 7};    // OK because so is C c = 7;
<int, C> c2 = {1, "foo"};  // Ill-formed because so is C c = "foo";
<int, C> c3({1, "foo"});          // OK because so is C c("foo");
<int, C> c4{1, "foo"};    // OK to be consistent with uniform-initialization
```

In contexts where template argument deduction applies, the `auto` specifier may be used as a parameter. This can be useful for ensuring one has a tuple type rather than an initializer list:

```
int a = 1, b = 2;
auto x0 = {a, nullptr}; // <int, nullptr_t>
auto x1 = {a, b}; // initializer_list<int>. Intended?
<auto, auto> x2 = {a, b}; // <int, int>. Intent clear
<auto...> x3 = {a, b};  // <int, int>
<integral auto ...> x4 = {a, b}; // <int, int>
<auto &...> x5 = {a, b};  // <int &, int &>
```

Note that empty tuples i.e. `<>` are perfectly valid (see use case below).

4.4 Empty tuples

Some people might be surprised that we allow empty tuples `<>` since they don't contain any types. However, we note that not only is `tuple<>` allowed, but it appears almost one thousand times in the ACTCD19 dataset. Here is an example of where it would come in handy in the `log` function presented earlier.

```
template<typename... Ts>
void log(string_view msg, <const Ts&...> args = <>, source_location sl =
source_location::current()) {
    cerr << sl << ":" << format(msg, ...args...);  //pack the tuple and expand it (more on that
below)
}
log("Message without args");            // Ok, args convert to <>
log("Message with two args", foo, bar); // Ok, 2 args
```

5. Tuple indexing and slicing

5.1 Tuple indexing by position or type

Although native tuple indexing can still be done via `std::get<>` (see below for a simple implementation), this paper proposes a new indexing operator `.<>` having the following signatures:

```
//indexing by position
template <size_t I> constexpr auto&& operator.<>();

//slicing (always returns a native tuple - see 5.2)
template <size_t...I> requires (sizeof...(I) > 1)
constexpr <auto&&...> operator.<>();

//indexing by type (see 5.3)
template <typename T> constexpr const T& operator.<>();
template <typename T> constexpr T&& operator.<>();
```

Both indexing by type and position will operate as expected:

```
auto t = {1, 2.2, "foo"}; //tuple of type <int, double, const char*>
auto i = t.<0>; //i = 1. decltype(t.<0>) is int&
auto& d = t.<double>; //d references 2nd member in t by type
d = 3.3; //now t == {1, 3.3, "foo"}
t.<3> = "bar"; //update t's 3rd member
```

5.2 Range indexing a.k.a tuple slicing

The indexing operator can be used to capture a tuple range (or slice). This can be accomplished by providing a comma-delimited set of indices inside the index operator. To specify a range, the `..` notation is introduced in between two indices. A more obvious choice would perhaps be the colon ':' but `<:` and `:>` happen to represent digraphs for `[` and `]` respectively, which rules them out. The following table outlines some slice expressions:

If we have a native tuple `t` of `x` elements and let `Tn` be the type at position `n`, then:

Notation	Representation
`t.<n>`	Reference to element as position `n`. Type == `Tn&`
`t.<-n>`	Reference to element as position `x-n-1`. Type == `T(x-n-1)&`
`t.<n..n+1>`	Slice of the element a position `n`: Type == `<T&>`

Notation	Representation
`t.<n1,n2,n3>`	Slice of 3 distinct elements: Type == `<Tn1&, Tn2&, Tn3&>`
`t.<..n>`	Slice of elements `0` to `n-1`; Type == `<T0&, T1&, ...T(x-1)&>`
`t.<n..>`	Slice of elements from position `n` to the end: Type == `<Tn&, T(n+1)&, ...T(x-1)&>`
`t.<..-n>`	Slice of all elements but the last `n` ones: Type == `<...T(x-n-2)&, T(x-n-1)&>`
`t.<..>`	Slice of all elements: Type == `<T0&,...,T(x-1)&>`
`t.<n1,n3..n8,n10>`	Slice mixing a range and individual indices of `t`

Note that for brevity purposes T& can be 'const T&', 'T&' or 'T&&' depending on t's access qualifiers

It is also worthwhile to note that slicing a tuple will always return a smaller or equal size tuple.

Consider:

```
<int, double, std::string, int> t;
auto t1 = t.<0..2>; //t1 == <int&, double&>
<auto&...> t2; //alternate notation to t1
t1.<0> = 5; //set the first element of t via indexed access.
t.<1..3> = {2.2, "foo"}; //set t.<1> and t.<2>
```

5.3 Getting the template type with the index operator

In order to get the type of a specific element within an `std::tuple`, the `std::tuple_element<size_t, T>` utility can be used for compile-time indexed access. Similarly, native tuples can leverage the `T::<size_t>` notation for compile-time access.

```
template <typename...Ts>
void foo(Ts&&...ts) {
    <Ts&&...>::<0> a; //a represents the 1st type in the pack
    auto&& b = {std::forward<Ts>(ts)...}.<0>; //initialized with the 1st value in the pack
}
```

5.4 Ellipsis `...` a.k.a. packing operator

There are instances when a tuple needs to be broken apart into its individual constituents. For example decomposing the tuple to invoke functions with variadic arguments or instantiating objects of variadic types. To use native tuples in these scenarios, a *tuple-to-pack* conversion operation must be applied. Once a tuple becomes a pack, it can be unpacked following current unpacking syntax, the ellipsis *suffix* `t...`. The current paper proposes mirroring the unpacking syntax but applied as a *prefix* operator instead: `...t`.

The `...` prefix blends itself well with the notion of packs as shown in the following examples where `t` has native tuple type `T`.

1. Variadic templates

```
template <typename ...Ts> struct X {};
template <typename ...Ts = ...t> struct Y {}; // Default value for variadic
```

2. Variadic lambda capture (since C++20) This notation also feels natural with C++20 variadic lambda captures.

```
[&...Ts = ...t](){ /* work with Ts... */ } //Ts is a pack captured by reference
```

There is a slight distiction however when it applies to native tuples. The `...Ts` declares a pack, whereas `...t` converts a tuple into a pack. One is a declaration, the other is an operation.

Usage scenarios:

1. Variadic type templates

```
<int, double> t = {1, 2.2};
template <typename ...Ts> struct S {};

//Instantiating S can be done in various ways:
S<int, double> s; //the natural way
S<...t...> s; //pack t then expand its types
S<...<int, double>...> s; //pack and expand an anonymous tuple
S<std::decay_t<...decltype(t.<..>)>...> s; //slicing, packing and unpacking a decayed t
```

2. Default variadic template notation.

```
//provide default arguments for a type pack, in this case an int and a double.
template <typename ...T = ...<int, double>> struct S{};

//Note that the following is ill-formed (expects a pack)
template <typename ...T = <int, double>> struct S{};
```

3. Getting a tuple size

```
auto t = {/*...*/};
auto size = sizeof...(...t); //tuple needs to be a pack
auto size = std::tuple_size<decltype(t)>::value; //use std utility
```

Why do we even need a packing operator for native tuples?

One question that comes to mind is why not allow native tuples to automatically expand e.g. `t...` and why require it to be converted into a pack before doing so? The answer is well explained in the chapter 4.2 of P1858.

For example purposes, we are reproducing the excerpt below:

```
template <typename TUPLE, typename... U>
void call_f(TUPLE t, U... us)
{
    // Is this intended to add 't' to each element in 'us'
    // or is intended to pairwise sum the tuple 't' and
    // the pack 'us'?
    f((t + us)...);
}
```

By allowing automatic tuple expansion using `...` notation, it becomes impossible to disambiguate between the two possible unpacking scenarios. Therefore a tuple must be packed *explicitly* before being unpacked.

Now the above notation can be expressed both ways depending on intent

```
f((t + us)...); //add 't' to each element of 'us'
f((...t + us)...); //pairwise sum of 't' and 'us'
```

5.6 Implementing `std::get`, `std::tuple_element` and `std::tuple_size`

Using the above notations, it becomes very easy to retrofit these std utilities to also work with native templates:

`std::get<>` example

```
template <std::size_t I, typename ...T>
requires (I < sizeof...(T))
constexpr <T...>::<I>& get(<T...>& t) noexcept { return t.<I>; };
```

`std::tuple_element` example

```
template<std::size_t I, typename T>
struct tuple_element; //base

template<std::size_t I, typename ...T>
struct tuple_element<I, <T...>>
{
    using type = <T...>::<I>;
};
```

`std::tuple_size` example

```
template <typename T>
struct tuple_size; //base

template <typename...T>
struct tuple_size<<T...>>
{
    static constexpr size_t value = sizeof...(T);
}
```

5.7 Interactions with P1858

Paper P1858 generalizes pack usage and proposes various new operators and syntax elements to the language, including an indexing and packing operator `.[:]`. The authors of this native tuple proposal have been coordinating with the author of P1858R2 with the goal of eventually converging on a majority of points. As these divergences are largely syntactic bikesheds, we are optimistic that a suitable convergence will be found.

Nonetheless, we have diverged at several places from the notation of P1858. The major divergences together with their rationales are as follows

- We use . to extract an element from a native tuple (or tuple-like type) where P1858R2 use .[x]. We prefer the angle brackets to make clear that `x` needs to be a valid template argument and by analogy with `std::get`. For example, our `t.<2>` acts like `get<2>(t)` and `t.<int>` acts like `get<int>(t)`. We are concerned that the `t.[i]` notation looks like it could accept a non-constexpr int and in our opinion `t.[int]` just looks strange
- We use `..` instead of `:` for slices. E.g. We use `t.<..4>` where P1858 uses `t.[:4]`. This change is necessitate by C++ digraphs as `<:` is an alternative operator for `[` (Thanks to Barry Revzin for pointing this out)!
- Our slicing operator returns a sub-native-tuple rather than a range. This seems natural for operating on native-tuples and will likely be essential if we ever allow named fields in native tuples as described

in the Future Directions section * We prefer `...` as our packing operator for the reasons given in section 5.4.

We illustrate these more concretely below:

If we have the following:

```
template <typename ...Ts>
struct S{
   S(Ts...ts);
   tuple<Ts...> tup;
};
```

P1858
`ts...[0:3]` : Produces a sub-pack pack of values
`tup.[:]...[0:3]` : Packs the tuple and produces a sub-pack of values.
`Ts...[0:3]` : Produces a sub-pack of types.
`tuple<Ts...>::[:]...[0:3]` : Produces a sub-pack of types.
Single notation for slicing and packing
Compile-time indexing

Previous example modified for native tuples:

```
template <typename ...Ts>
struct S{
   S(Ts...ts);
   <Ts...> tup;
};
```

P2163R0
`...tup` : Packs a native tuple.
`tup.<0..3>` : Slices the tuple object. This produces another tuple.
`<Ts...>::<0..3>` : Defines a native tuple composed of types 0 to 2.
Slicing and packing have two distinct notations
Compile-time indexing

As one can see, the differences are subtle but important. One potential resolution which could be up for discussion, would let P1858 could adopt the prefix `...` notation for packing instead of `[:]` and the slicing operator `.<..>` would produce an anonymous tuple when applied to a pack instead of producing a sub-pack. Another possibility would be to allow the slicing operator to return the same type that it's slicing. For now, we simply wanted to point out the similarities and salient differences between both proposals.

6. Usage in variadic template classes

Native tuples can easily be declared and used in variadic type templates. The following example demonstrates this.

```
template <typename ...Ts>
struct S{
   S(Ts...ts) {
       auto t = {ts...}; //Use initializer list with the expanded arguments
   }
    template <typename ...Us>
    void foo(<Us...>&& u) {
        <Ts..., Us...> u2; //combine two packs
    }
   <Ts...> tup; //as a class member
};
```

Another example would be participating in fold expressions.

```
template<typename ...Ts>
int sum(const <Ts...>& t) {
    return (...t + ... + 1); //sum-up all tuple values and add 1
}
```

7. Usage as return types and automatic tuple deduction

Returning multiple values from functions is another very useful application of native tuples. The current prescribed way of doing this is either packing data inside a `std::tuple` or passing *in-out* arguments into the function. With native tuples, the code becomes more expressive and easier to understand without the need to introduce another type such as `std::tuple` for the sole purpose of packing values.

Consider the following:

```
//express clearer intent
<int, double, string> foo()
{
```

```
    int i, double d, string s;
    //do something with local variables
    return {i,d,s}; //RVO applies
}

//use auto-deduction for simplicity
auto foo() {
    return {1, 2.2, "bar"};
}
```

8. Passing native tuples as function arguments

In order to pass a tuple into a function expecting discrete arguments, packing and unpacking the tuple is necessary.

```
<A, B, C, D> foo();
void bar(const A&, B, const C&, D);

//packs the return value from foo() and then expand it for the first 3 positional parameters.
bar(...foo()..., {});

//pass only the 2nd and 3rd parameter
bar({}, ...foo().<1..3>..., {});
```

9. Structured bindings

Using the previous example we can also extract values with structured bindings.

```
<A, B, C, D> foo();
auto [a,b,c,d] = foo(); //binds all 4 values from tuple
auto [a,b] = foo().<0..2>; //binds only the first 2 values
auto& [ra,rb,rc,rd] = foo(); //bind references from the expanded named pack
auto& [ra,rb,rd] = foo().<0,1,3>; //skip the 3rd binding
```

Interactions with P1858 can also produce this:

```
auto [...p] = foo(); //p is a named pack of <A, B, C, D>.
auto [a, ...rest] = foo(); //a binds to A, rest is a <B, C, D> pack
```

10. Default initializers for parameter packs

As mentioned above, this proposal helps address CWG1263 on default arguments for parameter packs. The basic idea is

```
template <typename ...T> void f(T ...t = ...{1, 2, 3});
f(); // f<int, int, int>(1, 2, 3)
f(1.2); // f<double>(1.2)
// To override the defaults with no arguments, state it explicitly
f(...{}...); // f<>() - only needed because the default was not {}
```

See the `log` example above for a practical use case.

11. Future directions

Future research includes the feasability of allowing named fields within a tuple declaration as well as in aggregate initializer lists. Although these seem like a viable extension of current aggregate initialization rules, and are certainly a useful addition, there are technical hurdles which need further inspections (see Hurdles below).

11.1 Designated members and default initialization

As mentioned earlier, allowing named fields in tuples by analogy with braced initializer lists is appealing.

```
<int a, double b> x1 = {.a = 2, .b = 3}; // x1.a = 2, x1.b = 3
auto x2 = {.a = 2, .b = 3}; // Same as previous
auto f() { return {.a = 2, .b = 3}; } // Would be possible
```

Default values for fields seem natural as well

```
<int, string = "foo"s> a = {4};
```

Notes If we are to allow designated members and initializers(or lack thereof), they *must* become part of the type. Consider the following use cases:

```
<int a, double b> t1;
<int aa, double b> t2;
<int aa = 5, double b> t3;
decltype(t1) != decltype(t2) != decltype(t3);
```

Not including member names and initializer values in the type can lead to subtle ambiguities such as this one:

```
using a1 = <int a = 1>;
using a2 = <int a = 2>;
template<typename T> int f(T, T) { return T(); }
f(a1{}, a2{}); //what does this return?
```

or

```
void f(<int i = 1, double, vector<int> v = {1,2,3}>);
f(<int i = 5, double, vector<int> v = {10,11}>()); //what tuple values does f() get called
with?
```

This would also allow preserving equivalent behavior which applies to classes and PODs:

```
auto t = std::make_unique<<int a = 1, double b = 2.2>>();
t.a == 1; //default values and member names are preserved
```

11.2 Aggregate list initalization of native tuples

If we are to allow designated member names in aggregate initialization lists, we would certainly apply the following rule set as an extension to the current rules:

- If a native tuple member has a name, the initialization list must either use the same name designator or none at all.
- If a native tuple member *does not* have a name, the initialization list cannot declare one.
- If both tuple declaration and intializer list provide a default initializer, the latter takes precedence. This behavior remains identical to default member initialization for `struct` and `class` types.

```
<int a, double b> t = {.a=1, .b=2.2}; //OK. Member names match
<int a = 1, double b = 2.2> t = {.a=2, .b=3.3}; //OK. Member names match and t is initialized
to {2, 3.3};
<int a, double b> t = {1, .b=2.2}; //Ill-formed. Member 'a' is not designated in the
initializer list
<int a, double b> t = {.b=2.2}; //OK, a is default initialized
<int, double b> t = {.b=2.2}; //OK only b member is named. 'int' type is anonymous and default-
initialized
<int a = 1, double b = 2.2> t; //OK. Named members are default-initialized in the declaration
<auto, auto> t = {.a=1, .b=2.2}; //Ill-formed - members have not been declared.
<auto, auto> t = {1, 2.2}; //OK. Both members are anonymous
<auto...> t = {.a=1, .b=2.2}; //Ill-formed - members have not been declared.
```

11.3 Tuple indexing by member

Having named fields, allows new ways of accessing the tuple's values.

```
auto t = {.a=1, .b=2.2, .c="foo"};
t.b = 3.3; //Modify t via it's member
<A aa, B bb> t2;
t2.bb = {...}; //Modify t2 using its template field names
```

11.4 Slicing would preserve member names

```
<int a, double b, string s> t;
auto t2 = t.<0..2>;
t2.a = 5; //set t2's first member
```

11.5 Hurdles

Named fields are not without their own set of technical difficulties, and unless these can be solved, we cannot fully propose them in this paper. For example,

```
<auto, auto> x = {.a = 2, .b = 3};
```

What is the first `auto` deduced as? `int a=2` or just `int`? It is not clear what the former would mean, and the latter seems almost deceptive.

As another example, consider the case of a partial template specialization on tuples:

```
template <typename T> struct S {};
template <typename ...Ts> struct S<<Ts...>> { /*...*/ };

S<<int a, double b>> s;
//What type does 'Ts' represent? Can we even infer to it via the 'typename' keyword?
//Is it an 'int' or 'int a'? This would require a new way of deducing types!

//By the same token would this become valid?
S<int a> s;
```

Consider a metafunction which reverses `<int a, int b>`. One would expect the result to be `<int b, int a>` not `<int, int>`. However this is precisely what happens.

```
template<typename T> struct reverse;
template<typename T, typename ...Us> struct reverse< <T, Us...> > {
  using type = <typename reverse<Us...>::type, T>;
};
```

Unless `Us...` somehow preserves field names by way of an extension to type deduction system, the reverse function would not give expected results.

We are continuing to work on what form these ideas might reasonably take (and would greatly welcome any input) but they are clearly not ready to propose at this time.