Collected Issues for Tuples

Motivation

This paper collects together resolutions for a number of library issues related to tuples. As such it addresses NB comments ... against the CD ballot, which call for outstanding library issues to be resolved. However, it should be noted that a number of these issues were raised since the CD so might be deemed beyond the strict reading of such comments.

By collecting all issues into a consolidation paper it is hoped that any wording collisions can be identified and resolved without further burdening the project editor.

Issues

Constexpr constructors

A tuple should be a literal type (c.f. Core) if all its elements are in turn literal types. Among other benefits, this allows such tuples to guarantee constant initialization prior to dynamic initialization at program startup, making them suitable to burn into ROMs.

The first part of the solution is fairly simply, adding the constexpr qualifier to the default constructor. So long as all elements support a constexpr default constructor the keyword is respected. If this cannot be supported it is silently ignored, so there should be no penalty for non-literal classes.

In order for the copy constructor to be a literal constructor it must be specified with = default. Again, constexpr is only respected when it can be.

The final constructor to support that makes this feature valuable is the per-element constructor taking a parameter pack of values. At this point there is a problem as constexpr constructors must pass their arguments by value, but this constructor passes by reference. Changing the signature would risk a measurable performance change for the non-constexpr case that cannot be justified. It is hoped that a Core issue can be opened to support pass-by-reference-to-const in constexpr functions and constructors. Without such, the recommendation is that this issue be closed NAD.

The main drawback to making these requirements on the suggested constructors is that it limits the potential of implementations to do something interesting in the constructor body. While clearly there is no requirement to do any work here, we remove the freedom of implementations to perform additional work there, for example tracking construction in a debugging implementation.

rvalue get

There are times it might be useful to move a single element out of a tuple, especially when the tuple is the result of a function return. As such, it would be useful to have a form of get that returns an rvalue when bound to an rvalue reference.

Cget to avoid casting

Much like the addition of cbegin/cend to containers, it is not unreasonable to want a const-qualified reference from a tuple to forward to another function. This can be achieved by creating suitable intermediate values, or with judicious use of const_cast. However, a simpler solution (for library users) would be the addition of an additional form of get that always returns a reference-to-const. The suggested name is cget.

Note that overloads for non-const reference or rvalue reference are not necessary, as all bind to the same signature.

complete tuple-like interface

A tuple-like class could be anything that supports the trio of tuple calls, tuple_size, tuple_element and get. The standard library already supports three tuple-like classes in tuple, pair and array.

It is recommended that the tuple-like 'concept' is documented ahead of the tuple class itself, and this 'concept' is then used to frame the APIs that accept generic tuples, such as converting constructors and tuple_cat.

A SFINAEable check for tuple-like could be merely to test for the presence of a tuple_size specialization. Clearly this would go further in a full concepts-enabled version of the language, but seems an appropriate and minimal test for the current project.

The important missing piece to make use of this API is the ability to deduce an appropriate tuple-type or parameter pack from arbitrary tuple-like types.

```
// looks like we need to support a level of recursion obtaining the pack
template<typename T, size_t Limit, unsigned int ... N>
struct impl_tuple_type {
  // Need to recurse with one additional element, unless sizeof...(N) == Limit
  typedef tuple<tuple_element<N,T>...> type;
};

template<typename T, unsigned int ... N>
struct impl_tuple_type {
  typedef tuple<tuple_element<N,T>...> type;
};

template<typename T>
```

```
struct tuple_type {
  typedef typename impl_type_type<T, ??? tuple_size<T>::value>::type type;
};
```

Construction from tuple-like

std::tuple should be constructible from any compatible tuple-like object. In addition to other tuples, this would include std::pair and std::array. The current library achieves this by providing additional converting constructors for generic pair and tuple while ignoring array. All those overloads could be replaced by a single constructor accepting tuple-like objects of the same length.

concatenation of tuple-like

While the result of a tuple-cat operation should always be a std::tuple, there is no reason to restrict it to accepting instances of std::tuple. Rather, any tuple-like type could be supported.

```
template<typename A, typename B>
typename flatten_tuple<A,B>::type tuple_cat( A const & a, B const & b);
```

concatenation an arbitrary number of tuples

The existing tuple_cat API is limited to 2 tuples, although it can be called multiple times to build larger tuples. Even this simple support is specified using 4 overloads. However, this combinatorial explosion of overloads if we look to extend to 3 or more tuples should remind us of std::function and the motivation for perfect forwarding. All 4 overloads could be replaced with a single variadic signature that accepts an arbitrary number of tuple-like objects. This is considerably more versatile in use, and actually simpler to specify!

```
template< typename ... TupleLikes >
computed_type tuple_cat( TupleLikes && ... tpls );
```

Tuple-like API to support cv/ref qualified types

The tuple APIs tuple_size and tuple_element do not support cv-qualified tuples, nor references to tuples. Users can construct these for themselves with additional metaprogramming using the type traits, but it should not be necessary.

Precise wording for swap

Check this has not already been applied.

Abstract the type producer in make_tuple and make_pair

The decay/ref_wrapper support is common with make_pair and should be abstracted into a common API. Repeating an algorithm in words is a recipe for trouble in the future, and users may want to use the same component when creating their own tuple-like abstractions.

Value initialization vs. triviality

Is this already applied? Agreed resolution is that tuple should value initialize each member in its default constructor, just like pair.

Redundant move-assign operator

Waiting for revised WP to see if issue remains

Scoped allocator interaction

Check with Pablo

Visitor API

Consistent polymorphic functor requirements as for visitor, NAD Future for TR2 along-side the visitor proposal. Handy for streaming.

```
template< typename Visitor, typename ... TupleLikeTypes >
void visit_tuple( Visitor && v, TupleLikeTypes&&... t);
```

Short-circuited predicate support

There are a small number of algorithms that could use a vistor-based approach with a predicate object to return a result, but returning as soon as a result is determined rather than strictly evaluating for each set of elements for each of the tuples. For instance, this is the basis for implementing operator== and operator<.

```
template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_all_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_any_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_none_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Predicate, typename T, typename U >

int tuple_test_predicate_3way( Predicate && v, T && t, U && u );
```

Typelist support

Suggestion that tuple serves as basis of portable typelist facility, as type passed around rather than values in metaprograms. What facilities are missing to make this truly a first-class metaprogramming primitive as well as value primitive?

Probably NAD future for TR2, where can be evaluated against a true typelist facility

Zip facility

We have tuple-cat, zipping tuples (and tuple-like types) is next missing API

Pack/unpack function arguments

TR2 or beyond. Library support for storing function arguments in a tuple, and unpacking them in a perfectly forwarded function call.

```
template< typename Callable, typename TupleLike >
see below invoke( Callable && fn, TupleLike && t );
```

Implementation Experience

All the features that are supported by currently available compilers have been tested, which is essentially everything but constexpr. A sample implementation will be available on the committee wiki.

Proposed Wording

[Need a recent version of working paper to annotate – post N2914]

20.4 Tuples [tuple]

1 This subclause describes the tuple library that provides a tuple type as the class template tuple that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the tuple. Consequently, tuples are heterogeneous, fixed-size collections of values.

2 **Header** <tuple> **synopsis**

```
namespace std {
// 20.4.1, class template tuple:
template <class... Types> class tuple;

// 20.4.1.3, tuple creation functions:
const unspecified ignore;

template <class... Types>
  tuple<VTypes...> make_tuple(Types&&...);
template<class... Types>
  tuple<Types&...> tie(Types&...);
template< typename ... TupleLikeTypes >
  see below  tuple_cat( TupleLikeTypes && ... tpls );
```

~~template <class... TTypes, class... UTypes>~~
~~tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, const tuple<UTypes...>&);~~
~~template <class... TTypes, class... UTypes>~~
~~tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, const tuple<UTypes...>&);~~
~~template <class... TTypes, class... UTypes>~~
~~tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, tuple<UTypes...>&&);~~
~~template <class... TTypes, class... UTypes>~~
~~tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, tuple<UTypes...>&&);~~

```
template< typename Callable, typename TupleLike >
see below invoke( Callable && fn, TupleLike && t );

template< typename Visitor, typename ... TupleLikeTypes >
```

```
void tuple_visit( Visitor && v, TupleLikeTypes&&... t);

template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_all_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_any_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_none_pass( Visitor && v, TupleLikeTypes&&... tpls );

template< typename Predicate, typename T, typename U >
int tuple_test_predicate_3way( Predicate && v, T && t, U && u );
```

```
// 20.4.1.4, tuple helper classes:
template <class T> class tuple_size; // undefined
template <class... Types> class tuple_size<tuple<Types...> >;
template <size_t I, class T> class tuple_element; // undefined
template <size_t I, class... Types> class tuple_element<I, tuple<Types...> >;

template<typename T>
  struct tuple_size<const T> : tuple_size<T> {};

template<typename T>
  struct tuple_size<volatile T> : tuple_size<T> {};

template<typename T>
  struct tuple_size<const volatile T> : tuple_size<T> {};

template<typename T>
  struct tuple_size<T &> : tuple_size<T> {};

template<typename T>
  struct tuple_size<T &&> : tuple_size<T> {};

// 20.4.1.5, element access:
template <size_t I, class... Types>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>&);
template <size_t I, class ... types>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&);

template<size_t I, typename T>
struct tuple_element<I, const T>
    : add_const<typename tuple_element<I, T>::type> {};

template<size_t I, typename T>
struct tuple_element<I, volatile T>
    : add_volatile<typename tuple_element<I, T>::type> {};

template<size_t I, typename T>
struct tuple_element<I, const volatile T>
    : add_cv<typename tuple_element<I, T>::type> {};

template<size_t I, typename T>
struct tuple_element<I, T &> : tuple_element<I, T> {};

template<size_t I, typename T>
struct tuple_element<I, T &&> : tuple_element<I, T> {};

// 20.4.1.6, relational operators:
template<class... TTypes, class... UTypes>
  bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
  bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);
```

```
template<class... TTypes, class... UTypes>
  bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
  bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
  bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
  bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);
}
```

20.4.1.x TupleLike types [tuple.requirements]

1 The library describes a standard set of requirements for *TupleLike* types, which are types that describe a fixed-length sequence of values, where each value may be of a different but fixed type.

2 Table X describes the requirements on tuple-like types within this library. T is a tuple-like type, I is an integral constant expression of type std::size_t and t is an object of type T.

Table X

Expression	Result	Pre-condition
tuple_size<T>::value	The number of elements in the sequence described by T.	
typename tuple_element<I,T>::type	The type of the Ith element in the sequence described by T.	0 <= I < tuple_size<T>::value
get<I>(t)	A reference to the Ith element in the sequence described by T. If t is an lvalue, then an lvalue reference is returned. If t is an rvalue then an rvalue reference is returned. If t is cv-qualified, then the referenced type shall be similarly cv-qualified.	0 <= I < tuple_size<T>::value

3 A *TupleVisitor* is a Callable type that defines a set of operator() overloads that can be called with each element of a given TupleLike object. A *TuplePredicate* is a TupleVisitor where each overload of operator() returns a result contextually convertible to bool. A *MultiTupleVisitor* is a Callable type that defined a set of operator() overloads that can be invoked with multiple arguments corresponding to the Ith element of each of a set up TupleLike types. A *MultiTuplePredicate* is a MultiTupleVisitor where each invocation of operator() yields a value contextually convertible to bool. [Note: - such overloads may be provided by specific signatures covering each type in the passed TupleLikeTypes, or by means of a member function template operator() that should work with any TupleLikeTypes. End note].

4 Table Y describes the requirements on *TupleVisitor* objects that can be used with some library operations on TupleLike types. Fn is TupleVisitor type, fn is an object of type Fn, T is a TupleLike type, t is an object of type T and I is an integral constant expression of type size_t. Likewise fm is a MultiTupleVisitor object of type FM, and u is a function parameter pack of TupleLikeTypes.

Table Y

Expression	Result	Pre-condition
fn(get<I>(t))		0 <= I < tuple_size<T>::value
fm(get<I>(u)...)		0 <= I < tuple_size<decltype(u)>::value

20.4.1 Class template `tuple` [tuple.tuple]

```
namespace std {
template <class... Types>
class tuple {
public:
  constexpr tuple();
```

constexpr tuple(const tuple&) = default;
tuple(tuple&&);
~~explicit tuple(const Types&...);~~
template <class... UTypes>
explicit tuple(UTypes&&...);
template<typename TupleLike>
tuple(TupleLike &&);
~~template <class... UTypes>~~
~~tuple(const tuple<UTypes...>&);~~
~~template <class... UTypes>~~
~~tuple(tuple<UTypes...>&&);~~
~~template <class U1, class U2>~~
~~tuple(const pair<U1, U2>&); *// iff sizeof...(Types) == 2*~~
~~template <class U1, class U2>~~
~~tuple(pair<U1, U2>&&); *// iff sizeof...(Types) == 2*~~

// allocator-extended constructors
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, tuple&&);
~~template <class Alloc>~~
~~tuple(allocator_arg_t, const Alloc& a, const Types&...)~~;
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, ~~const~~ UTypes&&...);
template< class Alloc, typename TupleLike>
tuple(allocator_arg_t, const Alloc& a, TupleLike &&);
~~template <class Alloc, class... UTypes>~~
~~tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);~~
~~template <class Alloc, class... UTypes>~~
~~tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);~~
~~template <class Alloc, class U1, class U2>~~
~~tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);~~
~~template <class Alloc, class U1, class U2>~~
~~tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);~~

tuple& operator=(const tuple&) = default;
template<typename TupleLike>
tuple& operator=(TupleLike &&);
~~tuple& operator=(tuple&&);~~
~~template <class... UTypes>~~
~~tuple& operator=(const tuple<UTypes...>&);~~
~~template <class... UTypes>~~
~~tuple& operator=(tuple<UTypes...>&&);~~
~~template <class U1, class U2>~~
~~tuple& operator=(const pair<U1, U2>&); *// iff sizeof...(Types) == 2*~~
~~template <class U1, class U2>~~
~~tuple& operator=(pair<U1, U2>&&); *// iff sizeof...(Types) == 2*~~
};

// allocator-related traits
template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;
template <class... Types>
struct constructible_with_allocator_prefix<tuple<Types...> >;
}

20.4.1.1 Tuple traits [tuple.traits]

template <class... Types, class Alloc>

struct uses_allocator<tuple<Types...>, Alloc> : true_type { };

Requires: Alloc shall be an Allocator (20.1.2).

1 [*Note:* Specialization of this trait informs other library components that tuple can be constructed with an allocator, even though it does not have a nested allocator_type. —*end note*]

template <class... Types>

struct constructible_with_allocator_prefix<tuple<Types...> >

: true_type { };

[*Note:* Specialization of this trait informs other library components that tuple can be constructed with an allocator prefix argument. —*end note*]

20.4.1.2 Construction [tuple.cnstr]

constexpr tuple();

2 *Requires:* Each type in Types shall be default constructible.

3 *Effects:* ~~Default~~ Value initializes each element.

~~explicit tuple(const Types&...);~~

~~4 *Requires:* Each type in Types shall be copy constructible.~~

~~5 *Effects:* Copy initializes each element with the value of the corresponding parameter.~~

template <class... UTypes>

explicit tuple(UTypes&&... u);

6 *Requires:* Each type in Types shall satisfy the requirements of MoveConstructible (Table 33) from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).

7 *Effects:* Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).

~~tuple(const tuple& u);~~

~~8 *Requires:* Each type in Types shall satisfy the requirements of CopyConstructible(Table 34).~~

~~9 *Effects:* Copy constructs each element of *this with the corresponding element of u.~~

tuple(tuple&& u);

10 *Requires:* Each type in Types shall shall satisfy the requirements of MoveConstructible (Table 33).

11 *Effects:* Move-constructs each element of *this with the corresponding element of u.

template<typename TupleLike>
 tuple(TupleLike && tpl);

x1 *Requires:* tuple_size<T>::value == sizeof...(Types). Each type in Types shall satisfy the requirements of MoveConstructible (Table 33) or from the corresponding element in tpl if tpl is passed as an rvalue, and CopyConstructible from the corresponding element otherwise.

X2 *Effects:* Initializes each element of *this with the corresponding element of tpl which is forwarded as an rvalue if tpl is passed as an rvalue, or an lvalue otherwise. [*Note:* enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. —*end note*]

~~template <class... UTypes> tuple(const tuple<UTypes...>& u);~~

~~12 *Requires:* Each type in Types shall be constructible from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).~~

~~13 *Effects:* Constructs each element of *this with the corresponding element of u.~~

~~14 [*Note:* enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. —*end note*]~~

~~template <class... UTypes> tuple(tuple<UTypes...>&& u);~~

~~15 *Requires:* Each type in Types shall shall satisfy the requirements of MoveConstructible (Table 33) from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).~~

~~16 *Effects:* Move-constructs each element of *this with the corresponding element of u. [*Note:* enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. —*end note*]~~

~~template <class U1, class U2> tuple(const pair<U1, U2>& u);~~

~~17 *Requires:* The first type in Types shall be constructible from U1 and the second type in Types shall be constructible from U2. sizeof...(Types) == 2.~~

~~18 *Effects:* Constructs the first element with u.first and the second element with u.second.~~

~~template <class U1, class U2> tuple(pair<U1, U2>&& u);~~

~~19 *Requires:* The first type in Types shall shall satisfy the requirements of MoveConstructible(Table 33) from U1 and the second type in Types shall be move-constructible from U2. sizeof...(Types) == 2.~~

~~20 *Effects:* Constructs the first element with std::move(u.first) and the second element with std::move(u.second).~~

~~tuple& operator=(const tuple& u);~~

~~21 *Requires:* Each type in Types shall be CopyAssignable (Table 36).~~

~~22 *Effects:* Assigns each element of u to the corresponding element of *this.~~

~~23 *Returns:* *this~~

~~tuple& operator=(tuple&& u);~~

~~24 *Requires:* Each type in Types shall shall satisfy the requirements of MoveAssignable (Table 35).~~

~~25 *Effects:* Move-assigns each element of u to the corresponding element of *this.~~

~~26 *Returns:* *this.~~

~~template <class... UTypes>~~

~~tuple& operator=(const tuple<UTypes...>& u);~~

~~27 *Requires:* Each type in Types shall be Assignable from the corresponding type in UTypes.~~

~~28 *Effects:* Assigns each element of u to the corresponding element of *this.~~

~~29 *Returns:* *this~~

~~template <class... UTypes>~~

~~tuple& operator=(tuple<UTypes...>&& u);~~

~~30 *Requires:* Each type in Types shall satisfy the requirements of MoveAssignable (Table 35) from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).~~

~~31 *Effects:* Move-assigns each element of u to the corresponding element of *this.~~

~~32 *Returns:* *this.~~

~~template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);~~

~~33 *Requires:* The first type in Types shall shall satisfy the requirements of MoveAssignable (Table 35) from U1 and the second type in Types shall shall satisfy the requirements of MoveAssignable (Table 35) from U2. sizeof...(Types) == 2.~~

~~34 *Effects:* Assigns u.first to the first element of *this and u.second to the second element of *this.~~

~~35 *Returns:* *this~~

~~36 [*Note:* There are rare conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type T and if the parameter passed to the constructor is not of type T, but rather a tuple type that is convertible to T. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it it possible to compare the “nesting depths” of the source and target tuples and decide to select the element-wise constructor if the source nesting depth is smaller than the target nesting-depth. This can be accomplished using an enable_if template or other tools for constrained templates. —*end note*]~~

~~template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);~~

~~37 *Requires:* The first type in Types shall be Assignable from U1 and the second type in Types shall be Assignable from U2. sizeof...(Types) == 2.~~

~~38 *Effects:* Assigns std::move(u.first) to the first element of *this and std::move(u.second) to the second element of *this.~~

~~39 *Returns:* *this.~~

template<typename TupleLike>
tuple(TupleLike && tpl);

x1 *Requires:* tuple_size<T>::value == sizeof...(Types). Each type in Types shall satisfy the requirements of MoveAssignable-(Table 35) or from the corresponding element in tpl if tpl is passed as an rvalue, and CopyAssignable-from the corresponding element otherwise.

x2 *Effects:* Assigns to each element of *this the corresponding element of tpl which is forwarded as an rvalue if tpl is passed as an rvalue, or an lvalue otherwise.

X3 *Returns:* *this.

template <class Alloc>

tuple(allocator_arg_t, const Alloc& a);

~~template <class Alloc>~~

~~tuple(allocator_arg_t, const Alloc& a, const Types&...);~~

template <class Alloc, class... UTypes>

tuple(allocator_arg_t, const Alloc& a, ~~const~~ UTypes&&...);

template <class Alloc>

tuple(allocator_arg_t, const Alloc& a, const tuple&);

template <class Alloc>

tuple(allocator_arg_t, const Alloc& a, tuple&&);

template <class Alloc, typename TupleLike >

tuple(allocator_arg_t, const Alloc& a, TupleLike &&);

~~template <class Alloc, class... UTypes>~~

~~tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);~~

~~template <class Alloc, class... UTypes>~~

~~tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);~~

~~template <class Alloc, class U1, class U2>~~

~~tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);~~

~~template <class Alloc, class U1, class U2>~~

~~tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);~~

40 *Requires:* Alloc shall be an Allocator (20.1.2).

41 *Effects:* Equivalent to the preceding constructors except that the allocator argument is passed conditionally to the constructor of each element. Each member is *allocator constructed* (20.7.2) with a.

20.4.1.3 Tuple creation functions [tuple.creation]

template<class... Types>

tuple<*VTypes*...> make_tuple(Types&&... t);

42 Let Ui be decay<Ti>::type for each Ti in Types. Then each Vi in VTypes is X& if Ui equals reference_wrapper<X>, otherwise Vi is Ui.

43 *Returns:* tuple<VTypes...>(std::forward<Types>(t)...).

44 [*Example:*

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

—*end example*]

template<class... Types}

tuple<Types&...> tie(Types&... t);

45 *Returns:* tuple<Types&>(t...). When an argument in t is ignore, assigning any value to the corresponding tuple element has no effect.

46 [*Example:* tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

—*end example*]

~~template <class... TTypes, class... UTypes>~~

~~tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, const tuple<UTypes...>& u);~~

~~47 *Requires:* All the types in TTypes shall be CopyConstructible (Table 34). All the types in UTypes shall~~

~~be CopyConstructible (Table 34).~~

~~48 *Returns:* A tuple object constructed by copy constructing its first sizeof...(TTypes) elements from the corresponding elements of t and copy constructing its last sizeof...(UTypes) elements from the corresponding elements of u.~~

~~template <class... TTypes, class... UTypes>~~

~~tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, const tuple<UTypes...>& u);~~

~~49 *Requires:* All the types in TTypes shall be MoveConstructible (Table 33). All the types in UTypes shall be CopyConstructible (Table 34).~~

~~50 *Returns:* A tuple object constructed by move constructing its first sizeof...(TTypes) elements from the corresponding elements of t and copy constructing its last sizeof...(UTypes) elements from the corresponding elements of u.~~

~~template <class... TTypes, class... UTypes>~~

~~tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, tuple<UTypes...>&& u);~~

~~51 *Requires:* All the types in TTypes shall be CopyConstructible (Table 34). All the types in UTypes shall be MoveConstructible (Table 33).~~

~~52 *Returns:* A tuple object constructed by copy constructing its first sizeof...(TTypes) elements from the corresponding elements of t and move constructing its last sizeof...(UTypes) elements from the corresponding elements of u.~~

~~template <class... TTypes, class... UTypes>~~

~~tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, tuple<UTypes...>&& u);~~

~~53 *Requires:* All the types in TTypes shall be MoveConstructible (Table 33). All the types in UTypes shall be MoveConstructible (Table 33).~~

~~54 *Returns:* A tuple object constructed by move constructing its first sizeof...(TTypes) elements from the corresponding elements of t and move constructing its last sizeof...(UTypes) elements from the corresponding elements of u.~~

template <class... TupleLikeTypes>
see below tuple_cat(TupleLikeTypes &&... tpls);

53 *Requires:* All the types in template parameter pack TupleLikeTypes shall be TupleLike types. The type of all elements in each TupleLikeType shall be MoveConstructible (Table 33) if the corresponding argument is passed as an rvalue, or CopyConstructible if passed as an lvalue.

54 *Returns:* A tuple object constructed by calling get<I>(tpls) for every valid I in turn on each TupleLike object tpls. [Note – the effect is to concatenate each succeeding tuple in the parameter pack tpls into one large tuple. Elements are not interleaved].

```
template< typename Callable, typename TupleLike >
see below invoke( Callable && fn, TupleLike && t );
```

1 *Requires:* TupleLike shall be TupleLike type (***new table***). Callable shall be a function-like type that can be called with arguments equivalent to each element of TupleLike.

2 *Returns:* The result of invoking fn with arguments list get<I>(forward<TupleLike>(t))... where I evaluates as an incrementing index from 0 <= I < tuple_size<TupleLike>::value..

```
template< typename Visitor, typename ... TupleLikeTypes >
void tuple_visit( Visitor && v, TupleLikeTypes&&... t);
```

1 *Requires:* All the types in template parameter pack TupleLikeTypes shall be TupleLike types, and tuple_size<TupleLikeTypes>::value shall be the same for all such types. Visitor shall be a MultiTupleVisitor type over the TupleLike type in parameter pack TupleLikeTypes.

2 *Effects:* For each value I in the range 0 <= I <= tuple_size<T>::value (where T is any of the TupleLike types in template parameter pack TupleLikes) invokes the MultiTupleVisitor v with each set of arguments produced by calling get<I>(std::forward<TupleLikeTypes>(t)) on each TupleLike type in function parameter pack t in turn.

```
template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_all_pass( Visitor && v, TupleLikeTypes&&... tpls );
```

1 *Requires:* All the types in template parameter pack TupleLikeTypes shall be TupleLike types, and tuple_size<TupleLikeTypes>::value shall be the same for all such types. Visitor shall be a MultiTuplePredicate type over the TupleLike types in parameter pack TupleLikeTypes.

2 *Effects:* For each value I in the range 0 <= I <= tuple_size<T>::value (where T is any of the TupleLike types in template parameter pack TupleLikes) invokes the MultiTuplePredicate v with each set of arguments produced by calling get<I>(std::forward<TupleLikeTypes>(t)) for each TupleLike type T in function parameter pack t in turn. Returns immediately if any of those calls evaluates to false.

3 *Returns:* true if invoking v for each set of arguments supplied by tpls is true. Otherwise returns false. Returns true If there are no invocations as either sizeof...(TupleLikeTypes) == 0, of tuple_size<T>::value == 0 for the TupleLike types.

```
template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_any_pass( Visitor && v, TupleLikeTypes&&... tpls );
```

1 *Requires:* All the types in template parameter pack TupleLikeTypes shall be TupleLike types, and tuple_size<TupleLikeTypes>::value shall be the same for all such types. Visitor shall be a MultiTuplePredicate type over the TupleLike types in parameter pack TupleLikeTypes.

2 *Effects:* For each value I in the range 0 <= I <= tuple_size<T>::value (where T is any of the TupleLike types in template parameter pack TupleLikes) invokes the MultiTuplePredicate v with each set of arguments produced by calling get<I>(std::forward<T>(t)) for each TupleLike type T in function parameter pack t in turn. Returns immediately if any of those calls evaluates to true.

3 *Returns:* true if invoking v for any set of arguments supplied by tpls is true. Otherwise returns false. Returns false If there are no invocations as either sizeof...(TupleLikeTypes) == 0, of tuple_size<T>::value == 0 for the TupleLike types.

```
template< typename Visitor, typename ... TupleLikeTypes >
bool tuple_if_none_pass( Visitor && v, TupleLikeTypes&&... tpls );
```

1 *Requires:* All the types in template parameter pack TupleLikeTypes shall be TupleLike types, and tuple_size<TupleLikeTypes>::value shall be the same for all such types. Visitor shall be a MultiTuplePredicate type over the TupleLike types in parameter pack TupleLikeTypes.

2 *Effects:* For each value I in the range 0 <= I <= tuple_size<T>::value (where T is any of the TupleLike types in template parameter pack TupleLikes) invokes the MultiTuplePredicate v with each set of arguments produced by calling get<I>(std::forward<T>(t)) for each TupleLike type T in function parameter pack t in turn. Returns immediately if any of those calls evaluates to true.

3 *Returns:* false if invoking v for any set of arguments supplied by tpls is true. Otherwise returns true. Returns true If there are no invocations as either sizeof...(TupleLikeTypes) == 0, of tuple_size<T>::value == 0 for the TupleLike types.

```
template< typename Predicate, typename T, typename U >
int tuple_test_predicate_3way( Predicate && v, T && t, U && u );
```

1 *Requires:* Types T and U shall be TupleLike types, and tuple_size<T>::value == tuple_size<U>::value. The type Predicate shall be a MultiTupleVisitor type over types T and U.

2 *Effects:* Invokes the predicate v for each pair of elements in T and U, in tuple index order. Arguments are passes as rvalues if the corresponding parameter t or u is passed as an rvalue, or lvalue otherwise.

3 *Returns:* the value of the first invocation of v that does not return 0. Returns 0 if all invocations return 0, or if tuple_size<T>::value == 0.

20.4.1.4 Tuple helper classes [tuple.helper]

```
template <class... Types>
class tuple_size<tuple<Types...> >
  : public integral_constant<size_t, sizeof...(Types)> { };

template <size_t I, class... Types>
class tuple_element<I, tuple<Types...> > {
public:
   typedef TI type;
};
```

55 *Requires:* I < sizeof...(Types). The program is ill-formed if I is out of bounds.

56 *Type:* TI is the type of the Ith element of Types, where indexing is zero-based.

20.4.1.5 Element access [tuple.elem]

```
template <size_t I, class... Types>
typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);
```

57 *Requires:* I < sizeof...(Types). The program is ill-formed if I is out of bounds.

58 *Returns:* An lvalue reference to the Ith element of t, where indexing is zero-based.

59 *Throws:* nothing.

```
template <size_t I, class... Types>
typename tuple_element<I, tuple<Types...> >::type&& get(tuple<Types...>&& t);
```

x1 *Requires:* I < sizeof...(Types). The program is ill-formed if I is out of bounds.

x2 *Returns:* An rvalue reference to the Ith element of t, where indexing is zero-based.

x3 *Throws:* nothing.

```
template <size_t I, class... Types>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);
template <size_t I, class... Types>
  typename tuple_element<I, tuple<Types...> >::type const& cget(const tuple<Types...>& t);
```

60 *Requires:* I < sizeof...(Types). The program is ill-formed if I is out of bounds.

61 *Returns:* A const reference to the Ith element of t, where indexing is zero-based.

62 *Throws:* nothing.

63 [*Note:* Constness is shallow. If a T in Types is some reference type X&, the return type is X&, not const X&. However, if the element type is non-reference type T, the return type is const T&. This is consistent with how constness is defined to work for member variables of reference type. —*end note*]]

64 [*Note:* The reason get is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the template keyword. —*end note*]

20.4.1.6 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

65 *Requires:* For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).

66 *Returns:* true iff get<i>(t) == get<i>(u) for all i. For any two zero-length tuples e and f, e == f returns true.

67 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

```
template<class... TTypes, class... UTypes>
bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Requires: For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) < get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).

68 *Returns:* The result of a lexicographical comparison between t and u. The result is defined as:

(bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t_{tail} < u_{tail}), where r_{tail} for some tuple r is a tuple containing all but the first element of r. For any two zero-length tuples e and f, e < f returns false.

```
template<class... TTypes, class... UTypes>
bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

69 *Returns:* !(t == u).

```
template<class... TTypes, class... UTypes>
bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

70 *Returns:* u < t.

```
template<class... TTypes, class... UTypes>
bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

71 *Returns:* !(u < t)

```
template<class... TTypes, class... UTypes>
bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

72 *Returns:* !(t < u)

73 [*Note:* The above definitions for comparison operators do not require t_{tail} (or u_{tail}) to be constructed. It may not even be possible, as t and u are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. —*end note*]