std::tuple

the std::tuple template

```
template <typename ... Types> class tuple {
public:
  template< typename ... Types2>
  tuple( Types2 && ...);
  /* black magic */
};
```

- example

```
using my_tuple = tuple< int, double, std::string>;
my_tuple t1( 1, 2.3, "Hello");
auto t2 = std::make_tuple(1, 2.3, std::string("Hello"));
my_tuple t3 = {1, 2.3, "Hello"};
```

- with C++17 deduction guides, the compiler can determine the template parameters
  - from the types of the arguments to the constructor
  - beware of implicit conversions

```
std::tuple t4(1, 2.3, std::string("Hello"));
auto t5 = std::tuple(1, 2.3, std::string("Hello"));
```

the std::tuple template

```
template <typename ... Types> class tuple /*...*/;
```

- element access: std::get

```
template < size_t I, typename ... Types>
  /*...???...*/ get( tuple< Types ...> & t);
template < typename T, typename ... Types>
  /*...???...*/ get( tuple< Types ...> & t);
```

- example

```
using my_tuple = tuple< int, double, std::string>;

my_tuple t1( 1, 2.3, "Hello");
```

- access by constant index

```
double v = std::get< 1>( t1);
std::get< 1>( t1) = 3.14;
```

- access by type - only if there is exactly one member of the type

```
double v2 = std::get< double>( t1);
```

the std::tuple template

- std::get is a global function
  - applicable to std::tuple, std::pair, std::array, std::complex, std::ranges::subrange
  - why not a member function?

```
template <typename ... Types> class tuple
{ public:
  template<std::size_t I>
  /*...???...*/ get();
}
```

  - a more complex example

```
template< typename TPL>
void print_something( TPL&& t)
{
  std::cout << t.get<1>();           // SYNTAX ERROR: Unexpected ")"
  std::cout << t.template get<1>();  // WOULD BE CORRECT IF get WAS A MEMBER
  std::cout << std::get<1>(t);       // CORRECT
}
```

the std::tuple template

```
template <typename ... Types> class tuple /*...*/;
```

- determining element type: tuple_element
  - this is a *traits* class

```
template < size_t I, typename T> struct tuple_element {
  using type = /* black magic */;
};
```

  - type alias for easier use

```
template < size_t I, typename T>
  using tuple_element_t = typename tuple_element< I, T>::type;
```

- example

```
using my_tuple = tuple< int, double, int>;
using alias_to_double = typename tuple_element< 1, my_tuple>::type;
using alias2_to_double = tuple_element_t< 1, my_tuple>;
```

the std::tuple template

```
template <typename ... Types> class tuple /*...*/;
```

- accessing an element

```
template <size_t I, typename ... Types>
  tuple_element_t< I, tuple< Types ...>> & get( tuple< Types ...> & t);
```

the std::tuple template – usage

- How do we iterate over the elements of a tuple?

```
template< typename T, typename F>
void for_each_element( T && t, F && f)
{
  static constexpr N = std::tuple_size_v<std::remove_reference_t<T>>;
  for ( std::size_t I = 0; I < N; ++ I)
    f( std::get< I>( t));
}
```

  - No! std::get< I> requires constant I
    - std::get< I> returns different types for different I

the std::tuple template – usage

- How do we iterate over the elements of a tuple?

```
template< typename ... Types, typename F>
void for_each_element( std::tuple< Types ...> & t, F && f)
{
  f( std::get< Types>( t)) ...;       // SYNTAX ERROR
}
```

  - std::get< T>(t) returns the element which has the type T from the tuple t
    - but it fails when T is present more than once
  - statement is not a correct place for pack expansion

the std::tuple template – usage

- How do we iterate over the elements of a tuple?

```
template< typename ... Types, typename F>
void for_each_element( std::tuple< Types ...> & t, F && f)
{
  sink( f( std::get< Types>( t)) ...);
}
```

  - historic trick: argument list may contain pack expansion

```
template< typename ... Types>
void sink( Types && ...) {}
```

  - but argument list does not ensure left-to-right order of evaluation
  - it still fails when T is present more than once

the std::tuple template – usage

- How do we iterate over the elements of a tuple?

```
template< typename ... Types, typename F>
void for_each_element( std::tuple< Types ...> & t, F && f)
{
   ( f( std::get< Types>( t)) , ... );
}
```

  - C++17 fold expression
    - expands to an expression with comma operator - left-to-right evaluation guaranteed
    - it still fails when T is present more than once

the std::tuple template – usage

- How do we iterate over the elements of a tuple?
  - std::get<T>(t) fails when T is present more than once
  - we must use std::get<I>(t) with an index
  - we need to generate the indices <0,...,sizeof...(Types)-1>

  - C++14 library contains this:

```
template< std::size_t ... IL>
using index_sequence = std::integer_sequence<std::size_t, IL...>;
```

    - Just an alias to a more general tag class

```
template< std::size_t N>
using make_index_sequence = /* black magic */;
```

    - The black magic ensures that make_index_sequence<N> == index_sequence< 0, 1, ..., N-1>
    - But what it is good for?

the std::tuple template – usage

- How do we iterate over the elements of a tuple?
  - C++14 ensures that
    make_index_sequence<N> == index_sequence< 0, 1, ..., N-1>

```
template< typename ... Types, typename F>
void for_each_element( std::tuple< Types ...> & t, F && f)
{
  helper( t, f, std::make_index_sequence< sizeof...( Types)>{});
}
```

  - The third argument to helper is an empty temporary object
    - Only the type of the object is referenced inside helper
    - Compilers will (probably) optimize the object out

```
template< typename T, typename F, typename ... Indexes>
void helper( T & t, F && f, std::index_sequence< Indexes ...>)
{
   ( f(std::get<Indexes>(t)) , ... );
}
```

the std::tuple template – details and explanation

- How to store the values in the tuple?

```
template <typename ... Types> class tuple : Types ... {/**/};
```

  - This will not work!
    - Non-class types may not be inherited
    - The same class may not be inherited twice

```
template <typename ... Types> class tuple : wrapper< Types> ... {/**/};
```

  - It does not solve the duplicity

```
template <typename ... Types> class tuple : wrapper< I, Types> ... {/**/};
```

  - Where do we get the index I?

- We mat need recursion!

the std::tuple template – details and explanation

- How to store the values in the tuple?
- Recursive solution:
  - Declaration

```
template <typename ... Types> class tuple;
```

  - Partial specialization – recursive inheritance

```
template <typename T0, typename ... Types> class tuple< T0, Types ...>
  : public tuple< Types ...>
{
  T0 v_;
};
```

  - Explicit specialization – stop recursion

```
template<> class tuple<> {};
```

the std::tuple template – details and explanation

- How to retrieve I-th element from a parameter pack?
- Recursion again!

  - Declaration

```
template <std::size_t I, typename ... Types> class get_ith;
```

  - Partial specialization – recursive inheritance

```
template <std::size_t I, typename T0, typename ... Types> class get_ith< T0, Types ...>
  : public get_ith< I-1, Types ...> {};
```

  - Partial specialization – stop recursion and "return a type"
    - This specialization has priority due to lower number of arguments

```
template <typename T0, typename ... Types> class get_ith< 0, T0, Types ...>
{ using type = T0; };
```

  - What happens if I >= sizeof...(Types) ?
    - No definition for get_ith< J> for J = I - sizeof...(Types)

the std::tuple template – details and explanation

- Tuple element does not receive a parameter pack!

```
using my_tuple = tuple< int, double, int>;

using alias_to_double = typename tuple_element< 1, my_tuple>::type;
```

- Use specialization
  - Declaration

```
template <std::size_t I, typename T> class tuple_element;
```

  - Partial specialization

```
template <std::size_t I, typename ... Types> class tuple_element< I, tuple<Types ...>>
  : public get_ith< I, Types ...> {};
```

  - tuple_element is also implemented for pair and array