Standard library traits and tags

std::iterator_traits

- Container-manipulation functions usually use iterators in their interface
- Such functions need to know some properties of the underlying containers
  - new versions of algorithms in std::ranges use concepts instead of std::iterator_traits

- If IT is an iterator type, **std::iterator_traits<IT>** contains the following types:
  - **difference_type** – a signed type large enough to hold distances between iterators
    - usually std::ptrdiff_t
  - **value_type** – the type of an element pointed to by the iterator
  - **reference** – a type acting as a reference to an element
    - this is the type actually returned by operator* of the iterator
    - usually value_type& or const value_type&
    - it may be a class simulating a reference (e.g. for vector<bool>)
  - **pointer** – a type acting as a pointer to an element
    - value_type*, const value_type*, or a class simulating a pointer
  - **iterator_category** – one of predefined tags describing the category of the iterator
    - std::input_iterator_tag, std::output_iterator_tag, std::forward_iterator_tag, std::bidirectional_iterator_tag, or std::random_access_iterator_tag
    - shall be used via template specialization or using std::is_same_v

std::iterator_traits

- Implemented in standard library as

```
template< typename IT> struct iterator_traits {
  using difference_type = typename IT::difference_type;
  using value_type = typename IT::value_type;
  using reference = typename IT::reference;
  using pointer = typename IT::pointer;
  using iterator_category = typename IT::iterator_category;
};
```

- Any class intended to act as an iterator shall define the five types referenced above
  - All the five are required by the <algorithm> library
  - For the <ranges> library, defining value_type and difference_type is sufficient
  - The five types shall be accessed only indirectly through std::iterator_traits

- Since raw pointers may act as iterators, there is a partial specialization:

```
template< typename T> struct iterator_traits<T*> {
  using difference_type = std::ptrdiff_t;
  using value_type = std::remove_cv_t<T>;
  using reference = T&;
  using pointer = T*;
  using iterator_category = std::random_access_iterator_tag;
};
```

  - std::remove_cv_t<T> removes any const/volatile modifiers from T

std::remove_cv_t

- The implementation of std::remove_cv_t<T>
  - Based on the traits template std::remove_cv<T>
    - general template

```
template< typename T> struct remove_cv { using type = T; };
```

    - partial specializations have higher priority if they match more precisely the actual argument

```
template< typename T> struct remove_cv< const T> { using type = T; };
template< typename T> struct remove_cv< volatile T> { using type = T; };
template< typename T> struct remove_cv< const volatile T> { using type = T; };
```

    - The result is represented by a member named “type” by convention, used directly as:

```
typename remove_cv<X>::type
```

  - For convenience, the result may be accessed using the type alias:

```
template< typename T> using remove_cv_t = typename remove_cv<T>::type;
```

    - “_t” suffix convention is widely used in std library
    - It can be used simply as:

```
remove_cv_t<X>
```

volatile

- volatile
  - Used to denote “non-memory” locations in address space (e.g. I/O ports)
    - Compilers never eliminate or reorder accesses to volatile locations
  - It is UNSUITABLE for communication between threads
    - A read from a volatile variable that is modified by another thread without synchronization or concurrent modification from two unsynchronized threads is undefined behavior due to a data race.
    - Use std::atomic<T> instead
  - Unless you program device drivers or embedded systems, you shall not use *volatile*
    - Nevertheless, your templates shall work even for volatile types
    - Always use std::remove_cv_t instead of std::remove_const_t

decltype() and std::remove_reference_t

- Technically, std::iterator_traits are no longer needed
  - It is still usually simpler to use them
- Replacing std::iterator_traits with decltype()

```
template< typename IT>
auto range_max(IT b, IT e) {
  using T = std::remove_cv_t<std::remove_reference_t<decltype(*b)>>;
  T m = std::numeric_limits<T>::lowest();
  for (; b != e; ++b)
    m = std::max(m, *b);
  return m;
}
```

- **decltype(E)** denotes the type of the expression E
  - More exactly: The return type declared for the outermost function invoked in E
  - This is the (compile-time) static type, see **typeid** for the (run-time) dynamic type
  - In the example, decltype(*b) denotes the return type of IT::operator*
    - This is usually T& or const T&
- decltype(E) must usually be used with **remove_reference_t** and **remove_cv_t**
  - const T& -> remove_reference_t -> const T -> remove_cv_t -> T
  - [C++20] remove_cvref_t<X> = remove_cv_t< remove_reference_t<X>>

decltype() and std::declval

- We use the ability of the compiler to infer the return type from the body

```
template< typename IT>
auto range_max(IT b, IT e) {
  using T = std::remove_cv_t<std::remove_reference_t<decltype(*b)>>;
  T m = std::numeric_limits<T>::lowest();
  for (; b != e; ++b)
    m = std::max(m, *b);
  return m;
}
```

- What if we wanted to specify the return type explicitly?
  - e.g., in a standalone declaration
  - Using the “auto f() -> T” syntax, we can reference the argument names

```
template< typename IT>
auto range_max(IT b, IT e) -> std::remove_cv_t<std::remove_reference_t<decltype(*b)>>;
```

  - Otherwise, we need std::declval<T>()
    - It creates an expression of type T from nothing (by casting nullptr to T*)

```
template< typename IT>
std::remove_cv_t<std::remove_reference_t<decltype(*std::declval<IT>())>>
  range_max(IT b, IT e);
```

    - std::declval is a library template function while decltype is a keyword

std::is_reference_v

- Traits returning constants, e.g. std::is_reference_v<T>
  - Based on the traits template std::is_reference<T>
    - general template

```
template< typename T> struct is_reference<T> : std::false_type {};
```

    - partial specializations have higher priority

```
template< typename T> struct is_reference<T&> : std::true_type {};
template< typename T> struct is_reference<T&&> : std::true_type {};
```

    - Uses two type aliases (logically acting as policy classes):

```
using false_type = std::integral_constant<bool, false>;
using true_type = std::integral_constant<bool, true>;
```

    - These are aliases of a particular case of a more general auxiliary class:

```
template< typename U, U v> struct integral_constant {
  static constexpr U value = v;
  // ... there are more members here ... explanation later
};
```

    - The result is represented by a static constexpr member named “value” by convention
  - For convenience, the result may be accessed using the global variable alias:

```
template< typename T> inline constexpr is_reference_v = is_reference<T>::value;
```

std::is_reference_v

- Use of (Boolean) constants in templates – important examples:

```
template< typename T> class example {
  static constexpr bool is_ref = std::is_reference_v< T>;
```

  - passing a constant to another template type (possibly specialized)

```
  using another_type = some_template< is_ref>;
```

  - std::conditional_t is a compile-time conditional expression acting on types:

```
  using my_type = std::conditional_t< is_ref,
    std::add_pointer_t< std::remove_reference_t< T>>,        // replace reference by pointer
    T>;

  void a_method() {
```

  - constexpr if
    - no runtime cost; the inactive branch is not semantically checked

```
    if constexpr (is_ref) { /*...*/ } else { /*...*/ }
```

  - passing a constant to a template function

```
    some_function< is_ref>();
```

  - passing a type representing a constant to a function via a runtime argument
    - it creates an object from the traits class (it shall no longer be called traits in this case)

```
    using is_ref_t = std::is_reference<T>;

    another_function( is_ref_t{});
  }
};
```

Value-less function arguments

- passing a type representing a constant to a function via a runtime argument

```
using is_ref_t = std::is_reference<T>;
another_function( is_ref_t{});
```

- an empty object is created from the traits class
  - no run-time value is passed through the argument (compilers usually produce no code for it)
- the argument is used to pass compile-time information, i.e. its type
- the function may be overloaded on the type
  - in the case of std::is_reference<T>, inheritance hierarchy also applies (this is slicing!)

```
void another_function( std::false_type) { /*...*/ }
void another_function( std::true_type) { /*...*/ }
```

- alternatively, the function may be a template

```
template< bool v> void another_function( std::integral_constant< bool, v>) {
  if constexpr (v) { /*...*/ } else { /*...*/ }
}
```

- Trick: std::integral_constant<T,v> also defines conversion operator to T returning v

```
template< typename X> void another_function( X a) {
  if constexpr (a) { /*...*/ } else { /*...*/ }
}
```

  - This allows defining the function as lambda:

```
auto another_function = [](auto a) { if constexpr (a) { /*...*/ } else { /*...*/ }; };
```

Tag arguments

- Distinguishing constructors
  - Another use-case for value-less function arguments
  - All constructors have the same name
    - the name cannot be used to specify the required behavior
  - Example: std::optional<T> can store T or nothing

```
using string_opt = std::optional< std::string>;
string_opt x;                                      // initialized as nothing
assert(!x.has_value());
string_opt y(std::in_place);                       // initialized as std::string()
assert(y.has_value() && (*y).empty());
string_opt z(std::in_place, "Hello");              // initialized as std::string("Hello")
assert(z.has_value() && *z == "Hello");
```

  - Implementation:

```
struct in_place_t {};                              // a tag class
inline constexpr in_place_t in_place;              // an empty variable of tag type
template< typename T> class optional { public:
  optional();                            // initialize as nothing
  template< typename... L>
  optional( in_place_t, L &&... l);      // initialize by constructing T from the arguments l
};
```

Tag arguments

- The same approach is also used for regular functions
  - The purpose is to have the same name for different implementations of the same functionality
- Example: std::for_each allows to select parallel execution:

```
std::for_each( std::execution::par, k.begin(), k.end(), [](auto && a){ ++a; });
```

  - std::execution::par is a global variable of type std::execution::parallel_policy
  - The parallel implementation of for_each:

```
template< typename IT, typename F>
void for_each( std::execution::parallel_policy, IT b, IT e, F f);
```

Tag arguments with parameters

- A tag class may carry a compile-time value
  - Example: The initialization of std::variant<T1,…,Tn>

```
using my_variant = std::variant< std::string, const char *>;
my_variant x( in_place_index<0>, “Hello”);    // initialized as std::string(“Hello”)
assert(x.index() == 0 && std::get<0>(x) == “Hello”);
my_variant y( in_place_index<1>, “Hello”);    // initialized as (const char *)(“Hello”)
assert(y.index() == 1 && !strcmp(std::get<1>(y), “Hello”));
```

  - Implementation:

```
template<std::size_t> struct in_place_index_t {};       // a tag class template
template<std::size_t I>
inline constexpr in_place_index_t<I> in_place_index;    // an empty variable of tag type

template< typename... TL> class variant { public:
  template< std::size_t I, typename... L>
  variant( in_place_index_t<I>, L &&... l);
  /*...*/
};
```

Employing type non-equivalence with tag classes

```
template< typename P>
class Value {
  double v;
  // ...
};

struct mass {};

struct energy {};

Value< mass> m;
Value< energy> e;

e = m;    // error
```

- Type non-equivalence
  - Two classes/structs/unions/enums are always considered different
    - even if they have the same contents
  - Two instances of the same template are considered different if their parameters are different
  - It also works with empty classes
    - Called **tag** classes
- Usage:
  - To distinguish types which represent different things using the same implementation
    - Physical units
    - Indexes to different arrays
    - Similar effect to *enum class*