Compile-time iteration

Compile-time iteration

- Example 1: print all arguments of a function
  - Using fold expression

```
template< typename ... TL>
void print( TL && ... vl)
{ (std::cout << ... << vl); }
```

- Example 2: print all elements of a tuple
  - Using fold expression

```
template< typename ... TL>
void print( const std::tuple< TL...> & t)
{ (std::cout << ... << std::get<TL>(t)); }
```

  - Problem: std::get with type arguments does not work for repeated types:

```
print( std::tuple<int,int>(1,2));
```

Compile-time iteration

- Example 2: print all elements of a tuple
  - Using fold expression

```
template< typename ... TL>
void print( const std::tuple< TL...> & t)
{ (std::cout << ... << std::get<TL>(t)); }
```

  - Problem: std::get with a type argument does not work for repeated types:

```
print( std::tuple<int,int>(1,2));
```

  - In addition, this print does not work for std::pair, even though std::get works

```
print( std::pair<int,int>(1,2));
```

  - This interface would be better:

```
template< typename T>
void print( T && t)
{ (std::cout << ... << std::get</*???*/>(t)); }
```

    - Problem: There is no variadic list in this context - we can't use fold expression

Compile-time iteration

- Example 2: print all elements of a tuple

```
template< typename T>
void print( T && t) {
  static constexpr auto n = std::tuple_size_v< std::remove_cvref_t< T>>;
  for (std::size_t i = 0; i < n; ++i)
    std::cout << std::get<i>(t);     // ERROR
}
```

  - Problem: std::get requires an index known at compile-time
    - Because its return type depends on the index
  - There is no "for constexpr"

Compile-time computations

- Generating a sequence of indexes:

```
std::make_index_sequence< N>
```

  - is an alias to the type

```
std::index_sequence< 0, 1, /*...*/, N-1>
```

- Problem: For a fold expression, we need a list

```
template< typename T>
void print( T && t) {
  static constexpr auto n = std::tuple_size_v< std::remove_cvref_t< T>>;
  using seq = std::make_index_sequence<N>;
  (std::cout << ... << std::get<seq>(t));     // ERROR, seq is not a variadic list
}
```

  - Type/constant lists exist only as formal template arguments

Compile-time computations

- Unpacking lists
  - By function template

```
template< typename T, std::size_t ... il>
void print_impl( T && t, std::index_sequence< il...>) {
  (std::cout << ... << std::get< il>(t));
};
```

    - The second (unnamed) argument is a tag, used to pass a type
    - The constant list il is unwrapped from the type by template argument deduction
  - Usage:

```
template< typename T>
void print( T && t) {
  static constexpr auto n = std::tuple_size_v< std::remove_cvref_t< T>>;
  using seq = std::make_index_sequence<N>;
  print_impl( std::forward<T>(t), seq{});
}
```

Compile-time computations

- Unpacking lists
  - By partial specialization

```
template< typename IS>
struct print_impl;
template< std::size_t ... il>
struct print_impl< std::index_sequence< il...>> {
   template< typename T>
   static void print( T && t)
   { (std::cout << ... << std::get< il>(t)); }
};
```

  - Usage

```
template< typename T>
void print( T && t) {
  static constexpr auto n = std::tuple_size_v< std::remove_cvref_t< T>>;
  using seq = std::make_index_sequence<N>;
  print_impl<seq>::print( std::forward<T>(t));
}
```

Compile-time computations

- Implementation of index sequences (a part of standard library, simplified)
  - Tag struct

```
template< std::size_t ... il> struct index_sequence {};
```

  - Recursive generator

```
template< std::size_t N, typename IS>
struct black_magic;
template< std::size_t N, std::size_t ... il>
struct black_magic< N, index_sequence< il...>>
: black_magic< N-1, index_sequence< 0, il+1...>>
{};
template< std::size_t ... il>
struct black_magic< 0, index_sequence< il...>>
{
  using type = index_sequence< il...>;
};
```

  - Wrapper

```
template< std::size_t N>
using make_index_sequence = typename black_magic< N, index_sequence<>>::type;
```

    - This implementation consumes $O(N^2)$ resources during compilation; more efficient ways exist

Compile-time iteration in general

Compile-time computations

- Loops in compile-time
  - Variadic templates often require iteration through elements
    - In simple cases, a fold-expression may be sufficient:

```
template<int ... al> constexpr int add_v = (0 + ... + al);
```

      - Fold-expressions always evaluate elements independently, then aggregate them
    - In more complex cases, recursion is required:

```
template<int ... al> struct max;

template<int a0> struct max<a0> { static constexpr int value = a0; }

template<int a0, int ... al> struct max< a0, al ...> {

  private: static constexpr int vl = max< al ...>::value;

  public: static constexpr int value = a0 > vl : a0 : vl;

};
```

Compile-time computations

- Loops in compile-time
  - Recursive iteration can process a polymorphic list
    - List elements:

```
template<int d> struct up {};

template<int d> struct right {};
```

    - List summation:

```
template<typename ... EL> struct sum2d;

template<> struct sum2d<> {

  static constexpr int value_x = 0;

  static constexpr int value_y = 0;

};

template<int d0, typename ... EL> struct sum2d< up<d0>, EL ...> {

  static constexpr int value_x = sum2d<EL ...>::value_x;

  static constexpr int value_y = sum2d<EL ...>::value_y + d0;

};

template<int d0, typename ... EL> struct sum2d< right<d0>, EL ...> {

  static constexpr int value_x = sum2d<EL ...>::value_x + d0;

  static constexpr int value_y = sum2d<EL ...>::value_y;

};
```

    - Usage:

```
using my_sum = sum2d< up<2>, right<3>, up<2>>;

static_assert( my_sum::value_y == 4);
```

Compile-time computations

- Variadic lists
  - Variadic lists exist only as arguments of templates
  - If a variadic list has to be "stored" or "returned", it must be wrapped
    - std::tuple may be a convenient wrapper for type lists
      - But declaring your own variadic tag class may be safer:

```
template<typename ... TL> struct type_list {};
```

    - std::integer_sequence does the same for lists of integers
  - Usage:

```
using my_list = type_list<up<2>, right<3>, up<2>>;

using my_int_list = std::integer_sequence< int, 2, 3, 2>;
```

  - Unwrapping is done by template specialization

```
template< typename L> struct list_sum2d;

template< typename ... EL> struct list_sum2d<type_list<EL ...>> : sum2d<EL ...> {};
```

      - Inheritance is (mis)used to copy the outputs of sum2d
    - Usage:

```
static_assert( list_sum2d<my_list>::value_y == 4);
```

Compile-time computations

- Iteration through wrapped lists may also be based on indexes
  - It requires additional features from the list wrapper:

```
template<typename L> static constexpr type_list_size = /*magic*/;

template<typename L, std::size_t i> using type_list_element = /*magic*/;
```

    - See std::tuple_size_v and std::tuple_element_t for the magic
  - Example – recursive pass-through using indexes:
    - Indexes are reversed to allow stopping at 0 by partial specialization

```
template<typename L, std::size_t rev_i> struct type_list_iterate_impl {

private:

  using iterate_rest = type_list_iterate_impl<L, rev_i – 1>;

  using element_i = type_list_element<L, type_list_size<L> - rev_i>;

public:

  static constexpr int value_x = extract_value_x<element_i> + iterate_rest::value_x;

  static constexpr int value_y = extract_value_y<element_i> + iterate_rest::value_y;

};
```

    - extract_value_x and extract_value_y may be implemented by partial specialization on the type of element_i
    - Partial specialization:

```
template<typename L> struct type_list_iterate_impl< L, 0> {

  static constexpr int value_x = 0;

  static constexpr int value_y = 0;

};
```

    - Public wrapper:

```
template<typename L> using type_list_iterate = type_list_iterate_impl< L, type_list_size<L>>;
```

Avoiding chaos in template arguments

- Template arguments declared with typename offer no clue on their purpose/format
  - Partially mitigated by aptly named templates

```
template<typename L> static constexpr type_list_size = /*magic*/;

template<typename L, std::size_t i> using type_list_element = /*magic*/;
```

- Concepts offer a chance to improve readability
  - And compile-time protection
- Method 1 - using a tag

```
struct type_list_tag {};
```

  - A concept to check for the tag

```
template<typename L> concept is_type_list = std::derived_from< L, type_list_tag>;
```

  - The original type modified to inherit from the tag

```
template<typename ... TL> struct type_list : type_list_tag {};
```

  - Templates accepting type_list as arguments

```
template<is_type_list L> static constexpr type_list_size = /*magic*/;

template<is_type_list L, std::size_t i> using type_list_element = /*magic*/;
```

Avoiding chaos in template arguments

- Concepts offer a chance to improve readability
  - Method 2 - detecting a particular type
  - The original type is not modified

```
template<typename ... TL> struct type_list : type_list_tag {};
```

  - A helper template

```
template<typename L> struct is_type_list_impl;

template<typename ... TL> struct is_type_list_impl< type_list< TL...>> : true_type {};
```

  - The concept

```
template<typename L> concept is_type_list = is_type_list_impl< L>::value;
```

  - Templates accepting type_list as arguments

```
template<is_type_list L> static constexpr type_list_size = /*magic*/;

template<is_type_list L, std::size_t i> using type_list_element = /*magic*/;
```

- Two interconnected named entities are always required
  - A concept
    - Often named using an adjective or verb - "integral", "invocable", "is_function"
    - But also using nouns, potentially colliding with types - "range", "input_iterator"
  - A type (or many such types) that satisfies the concept
    - Named using a noun - "function", "iterator"

Old-school compile-time evaluation vs. constexpr functions

Compile-time computations

- [C++11] constexpr functions
  - May be evaluated at compile-time
    - In the context of a *constant-expression* (e.g., a template parameter)
  - There are restrictions on code and data used in constexpr functions
    - the following applies to C++20; older versions had significantly stricter restrictions
    - prohibited code elements: goto/label, throw/catch, reinterpret_cast
      - new/delete allowed since C++20
    - prohibited data elements: types containing virtual functions or virtual inheritance
    - all functions (including constructors and destructors) invoked must be constexpr
    - only the following objects may be accessed:
      - constexpr variables
      - variables local to one of the constexpr functions involved
      - objects dynamically allocated and freed within the same constant-expression context
  - Problems:
    - constexpr functions became really usable in C++17
    - not all compilers implement all the features, especially of C++20
    - more complex functions may easily hit some internal limitation of the compiler

Compile-time computations

- Old-school constant expressions
  - No constexpr functions used
    - Only built-in operators on built-in types
  - Values "stored" only in constexpr variables or static data members
    - Many static data members may be generated using class-template instantiation
      - [C++14] global constexpr variables may be templated too
    - Logically, a templated constexpr variable/data member may be viewed as the result of a "function" invocation on the template arguments
      - This "function" is implemented by the initialization expression of the variable
      - Specialization of templates may allow further tricks
  - Example (practically useless):

```
template<int a, int b> constexpr int add_v = a + b;
```

    - used as:

```
std::array<int, add_v<10,20>> my_array;
```

    - The equivalent constexpr function is:

```
constexpr int add_f(int a, int b) { return a + b; }
```

    - used as:

```
std::array<int, add_f(10,20)> my_array;
```

Compile-time computations

- Old-school constant expressions
  - Another example (defunct):

```
template<int n> constexpr int fib_v = n < 2 ? 1 : (fib_v<n-1> + fib_v<n-2>);
```

    - It will cause infinite recursion because fib_v<n-1> and fib_v<n-2> are always instantiated, even if n < 2, because template instantiation precedes expression evaluation
    - The correct implementation is:

```
template<int n> struct fib {
  static constexpr int value = fib_v<n-1>::value + fib_v<n-2>::value;
};
template<> struct fib<0> { static constexpr int value = 1; };
template<> struct fib<1> { static constexpr int value = 1; };
```

    - Convenience wrapper:

```
template<int n> constexpr int fib_v = fib<n>::value;
```

    - The corresponding constexpr function:

```
constexpr int fib_f(int n) { return n < 2 ? 1 : (fib(n-1) + fib(n-2)); }
```

    - Surprise: The old-school implementation is significantly faster!
      - The template-instantiation system will cache the intermediate results, avoiding re-evaluation
      - It may be a motivation to use the old-school approach

Compile-time computations

- Old-school emulation of complex datatypes
  - Complex datatypes in constant-expression context must have constructors, i.e. constexpr functions – this is not old-school
  - Trick: Types may serve as compile-time values
    - A templated tag-class...

```
template<int a, int b> struct int_pair {};
```

    - ... may be used to represent a tuple of compile-time constants, e.g.:

```
template<int x> using neighborhood_range = int_pair<x-1,x+1>;
```

    - The individual constants may be retrieved using template specialization:

```
template<typename P> struct first;

template<int a, int b> struct first<int_pair<a,b>> { static constexpr int value = a; };
```

      - Convenience wrapper:

```
template<typename P> constexpr int first_v = first<P>::value;
```

    - The class may also be extended to allow extraction of values directly:

```
template<int a, int b> struct int_pair { static constexpr int value_a = a; /*etc.*/ };

template<typename P> constexpr int first_v = P::value_a;
```

Iterating through n-tuple elements

Example

- Working with tuple values

```
using my_triple = std::tuple< int, my_class, double>;

using my_transformed_triple = type_transform_t< my_triple, std::vector>;


my_triple a;

my_transformed_triple c;

auto my_value_function = /*...*/;

static_transform(a, c, my_value_function);
```

  - static_transform shall be an equivalent of std::transform for tuples
    - The intent is to apply f to every element of x and store the results in r
    - Implementation like this will **not** compile:

```
template< typename T1, typename T2, typename F>

void static_transform( T1 && x, T2 && r, F f)

{ for ( std::size_t i = 0; i < std::tuple_size_v< T1>; ++i)

    std::get<i>(r) = f( std::get<i>(x));

}
```

  - my_value_function is a polymorphic functor
    - applied to arguments of different types, may return different types
    - in this case, it transforms single values into vectors

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
  - The simplest case is a polymorphic lambda (C++14)

```
auto my_value_function = [](auto && x){
  return std::vector</*???*/>( 1, x);
};
```

  - Problem: How to derive the type for the vector?

```
std::vector<std::remove_cv_t<std::remove_reference_t<decltype(x)>>>
```

  - Problem: The argument x shall be passed using std::forward</*???*/>(x)
  - Since C++20, lambda may contain explicit template arguments

```
auto my_value_function = []<typename T>(T && x){
  return std::vector<std::remove_cvref_t<T>>( 1, std::forward<T>(x));
};
```

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
  - Explicitly defined functor

```
struct my_value_functor {
  template< typename T1>
  std::vector<T1> operator()( T1 && x) const
  { return std::vector<T1>( 1, std::forward<T1>(x));
  }
};
auto my_value_function = my_value_functor{};
```

  - BEWARE: It does NOT work correctly!
  - If the actual argument is an lvalue of type T, T1 would be T &
    - Reference collapsing rules apply
  - The result type would be a vector of references!

- The problem encountered here is quite frequent and often forgotten!

Example

- Polymorphic functors
  - applied to arguments of different types, may return diferent types
  - in this case, it transforms single values into vectors
  - Explicitly defined functor – corrected implementation

```
struct my_value_functor {
  template< typename T1>
  auto operator()( T1 && x) const
  { return std::vector<std::remove_cv_t<std::remove_reference_t<T1>>>
      ( 1, std::forward<T1>(x));
  }
};
```

  - std::remove_reference_t removes & or && if present
  - std::remove_cv_t removes const or volatile if present
  - C++20: std::remove_cvref_t combines the two
  - Using auto in the operator declaration avoids repeating the ugly type-id

Example

- Polymorphic functors
  - applied to arguments of different types, may return different types
  - in this case, it transforms single values into vectors
  - Polymorphic lambda – corrected implementation

```
auto my_value_function = [](auto && x){
  using x_t = decltype(x);
  using e_t = std::remove_cv_t< std::remove_reference_t< x_t>>;
  return std::vector<e_t>( 1, std::forward<x_t>(x));
};
```

  - decltype(x) is the type of x, always a (lvalue or rvalue) reference
  - std::forward is used differently from the usual practice, but it works:
    - Our code is equivalent to

```
template< typename U> auto f(U && x) { /*...*/ std::forward<U &&>(x) /*...*/ }
```

    - std::forward<T>(x) is by definition equivalent to static_cast<T&&>(x)
    - Therefore, std::forward<U&&> is equivalent to std::forward<U> due to reference collapsing

Example

- Iterating through n-tuple elements
  - A non-working implementation

```
template< typename A, typename R, typename F>
void static_transform(A && a, R && r, F f)
{
    using atuple = std::remove_cv_t< std::remove_reference_t< A>>;
    using rtuple = std::remove_cv_t< std::remove_reference_t< R>>;
    static constexpr auto asize = std::tuple_size_v< atuple>;
    static constexpr auto rsize = std::tuple_size_v< rtuple>;
    static_assert(asize == rsize, "tuples must be of equal length");

    for ( std::size_t i = 0; i < std::tuple_size_v< T1>; ++i)
        std::get<i>(r) = f( std::get<i>(x));
}
```

  - std::get<i> requires constant i
  - The compiler must be able to instantiate f() for the correct types

Example

- Iterating through n-tuple elements
  - Reformulated using a compile-time version of std::for_each
  - static_for_each_index calls transform_ftor for each index in the given range

```
template< typename A, typename R, typename F>
void static_transform(A && a, R && r, F f)
{
    using atuple = std::remove_cv_t< std::remove_reference_t< A>>;
    using rtuple = std::remove_cv_t< std::remove_reference_t< R>>;
    static constexpr auto asize = std::tuple_size_v< atuple>;
    static constexpr auto rsize = std::tuple_size_v< rtuple>;
    static_assert(asize == rsize, "tuples must be of equal length");

    static_for_each_index< 0, rsize>(
        transform_ftor< A, R, F>(
            std::forward< A>(a), std::forward< R>(r), std::move(f)));
}
```

Example

- static_for_each_index - interface

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f);
```

  - Logically, it shall call f(i) for every i in [b,e)
  - Problem: i must be a constant inside f
    - The compiler must create a separate instantiation for every f(i)
  - Problem: f<i>() does not work because f is a variable, not a function
    - f.operator()<i>() does not work either
  - Possible solution: call a normal member function instead of operator()

```
f.call<i>();
```

    - Requires a functor like

```
struct ftor {
    template< std::size_t i>
    void call() const
    { /*...*/ }
};
```

Example

- static_for_each_index - interface

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f);
```

  - Logically, it shall call f(i) for every i in [b,e)
  - Problem: i must be a constant inside f
  - A better solution: pass the constant through the type of an argument

```
f(static_index<i>{});
```

  - The argument type is an empty tag class

```
template< std::size_t i> struct static_index {};
```

  - Requires a functor like

```
struct ftor {
    template< std::size_t i>
    void operator()(static_index<i>) const
    { /*...*/ }
};
```

Example

- A better solution: pass the constant through the type of an argument
  - We can also use std::integral_constant as the tag class

```
template< std::size_t i>
using static_index = std::integral_constant<std::size_t, i>;
```

  - With integral constant, we may even use a (C++14) lambda:

```
auto ftor = [](auto tag_value){
  constexpr std::size_t i = tag_value;
};
```

  - std::integral_constant defines a conversion operator to std::size_t which, although not declared static, does not access the object and returns the template argument i
  - therefore, the member function may be evaluated in constant context although tag_value is a run-time object
  - at run-time, the tag_value is an empty object and most compilers will not reserve any space for it, consequently, there will be no run-time cost for dealing with this object
  - tag objects shall always be passed by value – passing by reference may have a run-time cost

Example

- A functor for static_transform

```
template< typename A, typename R, typename F> struct transform_ftor
{
    transform_ftor(A && a, R && r, F f)
    : a_(std::forward<A>(a)), r_(std::forward<R>(r)), f_(std::move(f))
    {}
    template< std::size_t i> void operator()(static_index<i>) const
    {   std::get< i>(std::forward<R>(r_)) = f_( std::get< i>(std::forward<A>(a_)));
    }
private: A && a_; R && r_; F f_;
};
```

- Formally, A && and R && in the constructor are NOT universal references
  - However, we will supply the type arguments A, R from a context where A && and R && are universal references (and consistent with the actual arguments to the constructor)
  - Therefore, the use of forward in the constructor is correct
  - Since the member variables a_ and r_ have the same type as the arguments, they also act as if they were universal references
  - The use of forward in operator() is correct because it is invoked only once for each i
- if A && is rvalue reference, f_ may steal from it
  - std::get propagates the rvalue/lvalue distinction from its argument

Example

- Static for-loop – an implementation by recursion
  - A class/struct template is required because we need partial specialization

```
template< std::size_t b, std::size_t n> struct for_each_index_helper {
    template< typename F> static void call(F && f)
    {   f(static_index< b>());
        for_each_index_helper< b + 1, n - 1>::call( std::forward<F>(f));
    }
};
```

  - Partial specialization stops the recursion

```
template< std::size_t b> struct for_each_index_helper< b, 0> {
    template< typename F> static void call(F &&) {}
};
```

  - Public wrapper

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F f)
{   for_each_index_helper< b, e - b>::call(std::move(f));
}
```

  - The public function receives the functor f by value, similarly to all std algorithms
  - Internally, we pass it by universal reference to avoid excess copying

Example

- Static for-loop – implementation without recursion
  - we need to generate the indices {0,...,n-1}
  - C++14 library contains this:

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

    - index_sequence is 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>

- Usage:
  - Use make_index_sequence<N> to generate a list of indices
    - The list is not accessible directly
  - Use partial specialization (or a function template) as a context in which the list is visible as a template parameter pack
    - Similar to the implementation of type_transform; instead of a type list argument of a tuple, here we have a constant list argument of an index_sequence

Example

- Static for-loop – implementation without recursion
  - A helper class

```
template< std::size_t b, typename S>
struct for_each_index_helper2;
```

  - A partial specialization is used to access the template parameter pack L

```
template< std::size_t b, std::size_t ... L>
struct for_each_index_helper2< b, std::index_sequence< L ...> >
{   template< typename F> static void call(F & f)
    {   (f( static_index< b + L>()), ...);
    }
};
```

  - A public wrapper

```
template< std::size_t b, std::size_t e, typename F>
void static_for_each_index(F && f)
{
    for_each_index_helper2< b, std::make_index_sequence< e - b> >::call(f);
}
```

Example

- Implementation of make_index_sequence
  - A tag struct template

```
template< typename T, T ... IL> struct integer_sequence {};
```

  - A helper struct to recursively generate a list of integers
    - Defines member type = integer_sequence< T, 0, 1, /*...*/, N-1, (IL+N) ...>

```
template< typename T, T N, T ... IL> struct integer_sequence_generator
: integer_sequence_generator< T, N-1, 0, (IL+1)...> {};
```

  - A partial specialization stops the recursion

```
template< typename T, T ... IL>
struct integer_sequence_generator< T, 0, IL...> {
    using type = integer_sequence< T, IL...>;
};
```

  - The public wrapper

```
template< std::size_t N>
using make_index_sequence = typename integer_sequence_generator< std::size_t, N>::type;
```

- This is recursive at compile time, allowing to avoid recursion at run time
  - The compile-time complexity (N^2) may be improved to (N*log(N))
    - Hint: <IL..., (IL+sizeof...(IL))...> doubles the length of the sequence