Compile-time iteration

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>>;
```

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>
```

- which may be unpacked by partial specialization:

```
template< typename IS> struct iterate_impl;

template< std::size_t ... il> struct iterate_impl< std::index_sequence< il...>> {

   static constexpr int value = (0 + ... + extract_value< il>);

};
```

- wrapped as

```
template< std::size_t N>

constexpr int iterate_v = iterate_impl< std::make_index_sequence< N>>::value;
```

- Note: Additional arguments will be needed to pass a type list to the extract_value template
- Note: Using the index-sequence is usually advantageous if fold expressions are used

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;
```